

**Oracle® XML Publisher**

Core Components Guide

Release 5.6.3

**Part No. E05078-01**

April 2007

Oracle XML Publisher Core Components Guide, Release 5.6.3

Part No. E05078-01

Copyright © 2006, 2007, Oracle. All rights reserved.

Primary Author: Leslie Studdard

Contributing Author: Tim Dexter, Klaus Fabian, Hiro Inami, Edward Jiang, Incheol Kang, Kazuko Kawahara, Kei Saito, Elise Tung-Loo, Jackie Yeung

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

#### U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

---

# Contents

**Send Us Your Comments**

**Preface**

## **1 Introduction**

XML Publisher Overview.....	1-1
-----------------------------	-----

## **2 Creating an RTF Template**

<b>Introduction</b> .....	2-1
Supported Modes.....	2-1
Prerequisites.....	2-2
<b>Overview</b> .....	2-2
Using the Business Intelligence Publisher Template Builder for Microsoft Word Add-in .....	2-2
Associating the XML Data to the Template Layout.....	2-3
<b>Designing the Template Layout</b> .....	2-7
<b>Adding Markup to the Template Layout</b> .....	2-7
Creating Placeholders.....	2-8
Defining Groups.....	2-12
<b>Defining Headers and Footers</b> .....	2-15
Native Support.....	2-15
<b>Images and Charts</b> .....	2-17
Images.....	2-17
Chart Support.....	2-19
<b>Drawing, Shape and Clip Art Support</b> .....	2-30
<b>Supported Native Formatting Features</b> .....	2-41
General Features.....	2-41

Alignment.....	2-42
Tables.....	2-42
Date Fields.....	2-45
Multicolumn Page Support.....	2-45
Background and Watermark Support.....	2-47
<b>Template Features</b> .....	2-49
Page Breaks.....	2-49
Initial Page Number.....	2-50
Last Page Only Content .....	2-51
End on Even or End on Odd Page.....	2-54
Hyperlinks.....	2-54
Table of Contents.....	2-57
Generating Bookmarks in PDF Output.....	2-57
Check Boxes.....	2-58
Drop Down Lists.....	2-59
<b>Conditional Formatting</b> .....	2-62
If Statements.....	2-63
If Statements in Boilerplate Text.....	2-63
If-then-Else Statements.....	2-64
Choose Statements.....	2-65
Column Formatting.....	2-66
Row Formatting.....	2-69
Cell Highlighting.....	2-71
<b>Page-Level Calculations</b> .....	2-73
Displaying Page Totals.....	2-73
Brought Forward/Carried Forward Totals.....	2-75
Running Totals.....	2-79
<b>Data Handling</b> .....	2-81
Sorting.....	2-81
Checking for Nulls.....	2-81
Regrouping the XML Data.....	2-82
<b>Using Variables</b> .....	2-88
<b>Defining Parameters</b> .....	2-89
<b>Setting Properties</b> .....	2-91
<b>Advanced Report Layouts</b> .....	2-93
Batch Reports.....	2-93
Cross-Tab Support.....	2-95
Dynamic Data Columns.....	2-98
<b>Number and Date Formatting</b> .....	2-102
<b>Calendar and Time Zone Support</b> .....	2-116
<b>Using External Fonts</b> .....	2-117

Advanced Barcode Formatting.....	2-118
<b>Advanced Design Options.....</b>	<b>2-119</b>
XPath Overview.....	2-119
Namespace Support.....	2-122
Using the Context Commands.....	2-123
Using XSL Elements.....	2-125
Using FO Elements.....	2-128
 <b>3 Creating a PDF Template</b>	
<b>PDF Template Overview.....</b>	<b>3-1</b>
Supported Modes.....	3-1
<b>Designing the Layout .....</b>	<b>3-2</b>
<b>Adding Markup to the Template Layout.....</b>	<b>3-3</b>
Creating a Placeholder.....	3-4
Defining Groups of Repeating Fields.....	3-7
<b>Adding Page Numbers and Page Breaks.....</b>	<b>3-9</b>
<b>Performing Calculations.....</b>	<b>3-13</b>
<b>Completed PDF Template.....</b>	<b>3-13</b>
<b>Runtime Behavior.....</b>	<b>3-14</b>
<b>Creating a Template from a Third-Party PDF.....</b>	<b>3-16</b>
 <b>4 Creating an eText Template</b>	
<b>Introduction.....</b>	<b>4-1</b>
<b>Outbound eText Templates.....</b>	<b>4-2</b>
Structure of eText Templates.....	4-2
Constructing the Data Tables.....	4-6
Command Rows.....	4-6
Structure of the Data Rows.....	4-12
Setup Command Tables.....	4-16
Expressions, Control Structures, and Functions.....	4-27
Identifiers, Operators, and Literals.....	4-30
 <b>5 XSL, SQL, and XSL-FO Support for RTF Templates</b>	
Extended SQL and XSL Functions.....	5-1
XSL Equivalents.....	5-6
Using FO Elements.....	5-7
 <b>6 Adding Template Translations</b>	
Translatable Templates.....	6-1

## **7 Setting Runtime Properties**

Setting Properties in a Configuration File.....	7-1
Structure.....	7-3
Properties.....	7-3
List of Properties.....	7-4
Font Definitions.....	7-14
Locales.....	7-16
Font Fallback Logic.....	7-17
Predefined Fonts.....	7-18

## **8 Using the XML Publisher APIs**

Introduction.....	8-1
XML Publisher Core APIs.....	8-1
PDF Form Processing Engine.....	8-2
RTF Processor Engine.....	8-8
FO Processor Engine.....	8-9
PDF Document Merger.....	8-20
PDF Book Binder Processor.....	8-27
Document Processor Engine.....	8-30
Bursting Engine.....	8-43
XML Publisher Properties.....	8-54
Advanced Barcode Font Formatting Implementation.....	8-58

## **A Supported XSL-FO Elements**

Supported XSL-FO Elements.....	A-1
--------------------------------	-----

## **Index**

---

# Send Us Your Comments

## Oracle XML Publisher Core Components Guide, Release 5.6.3

Part No. E05078-01

Oracle welcomes customers' comments and suggestions on the quality and usefulness of this document. Your feedback is important, and helps us to best meet your needs as a user of our products. For example:

- Are the implementation steps correct and complete?
- Did you understand the context of the procedures?
- Did you find any errors in the information?
- Does the structure of the information help you with your tasks?
- Do you need different information or graphics? If so, where, and in what format?
- Are the examples correct? Do you need more examples?

If you find any errors or have any other suggestions for improvement, then please tell us your name, the name of the company who has licensed our products, the title and part number of the documentation and the chapter, section, and page number (if available).

Note: Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the new Applications Release Online Documentation CD available on Oracle MetaLink and [www.oracle.com](http://www.oracle.com). It contains the most current Documentation Library plus all documents revised or released recently.

Send your comments to us using the electronic mail address: [appsdoc\\_us@oracle.com](mailto:appsdoc_us@oracle.com)

Please give your name, address, electronic mail address, and telephone number (optional).

If you need assistance with Oracle software, then please contact your support representative or Oracle Support Services.

If you require training or instruction in using Oracle software, then please contact your Oracle local office and inquire about our Oracle University offerings. A list of Oracle offices is available on our Web site at [www.oracle.com](http://www.oracle.com).





---

# Preface

## Intended Audience

Welcome to Release 5.6.3 of the *Oracle XML Publisher Core Components Guide*.

This book is intended to be a reference for use with the JD Edwards EnterpriseOne and PeopleSoft Enterprise implementations of Oracle XML Publisher. This guide contains information both for the business user on creating templates and for the implementor on setting configuration properties.

XML Publisher is based on the XSL-FO standard. Although it is not necessary, it may be helpful to have an XSL guide to use as companion to this book.

See Related Information Sources on page x for more Oracle product information.

## TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

## Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at <http://www.oracle.com/accessibility/>.

## **Accessibility of Code Examples in Documentation**

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

## **Accessibility of Links to External Web Sites in Documentation**

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

## **Structure**

- 1 Introduction**
- 2 Creating an RTF Template**
- 3 Creating a PDF Template**
- 4 Creating an eText Template**
- 5 XSL, SQL, and XSL-FO Support for RTF Templates**
- 6 Adding Template Translations**
- 7 Setting Runtime Properties**
- 8 Using the XML Publisher APIs**
- A Supported XSL-FO Elements**

## **Related Information Sources**

### **Oracle XML Publisher for PeopleSoft Enterprise**

This guide describes the features and procedures specific to the PeopleSoft Enterprise implementation of XML Publisher.

### **Oracle XML Publisher for JD Edwards EnterpriseOne**

This guide describes the features and procedures specific to the JD Edwards EnterpriseOne implementation of XML Publisher.

---

# Introduction

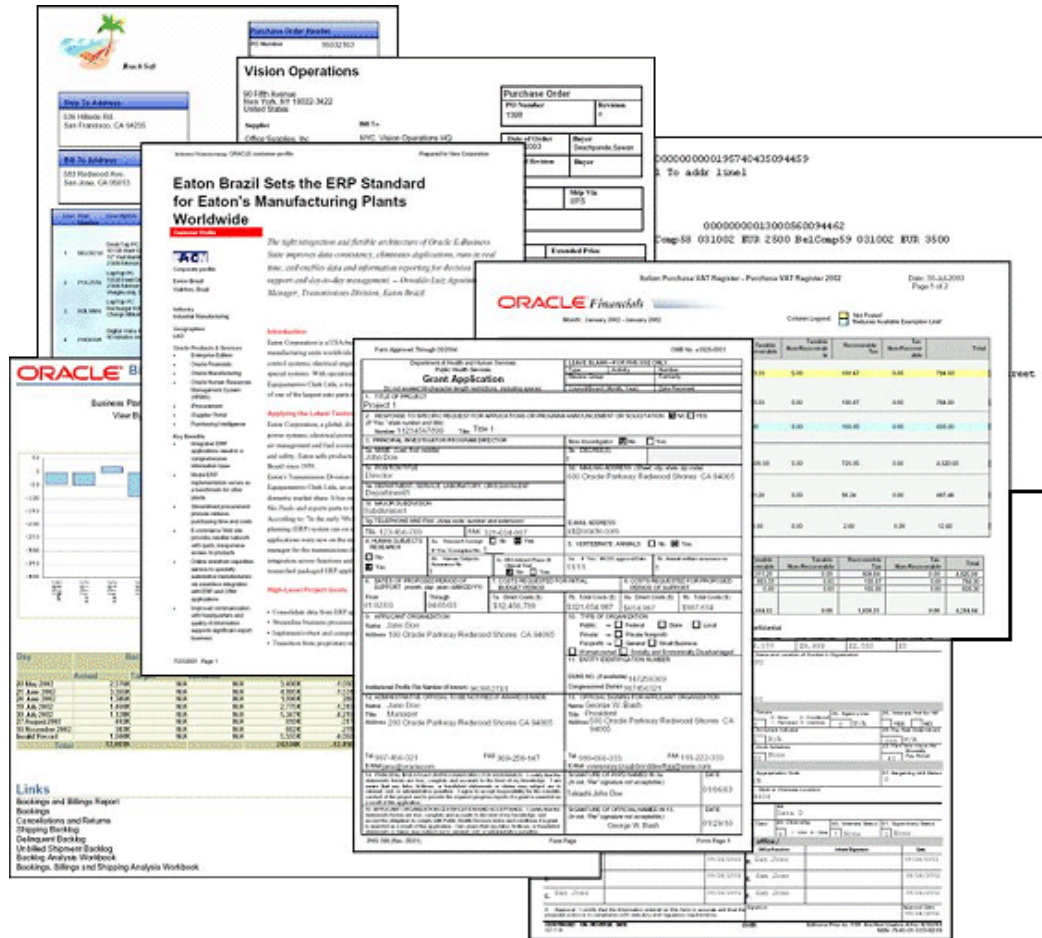
## XML Publisher Overview

Oracle XML Publisher is a template-based publishing solution delivered with the Oracle E-Business Suite, Peoplesoft Enterprise, and JD Edwards EnterpriseOne. It provides a flexible and robust approach to report design and publishing by integrating familiar desktop word processing tools with existing data reporting. XML Publisher leverages standard, well-known technologies and tools, so you can rapidly develop and maintain custom report formats.

The flexibility is a result of the separation of the presentation of the report from its data structure. The collection of the data can still be handled by existing report tools, but with XML Publisher you can design and control how the report outputs will be presented in separate template files. At runtime, your designed template files are merged with the report data to create a variety of outputs to meet a variety of business needs, including:

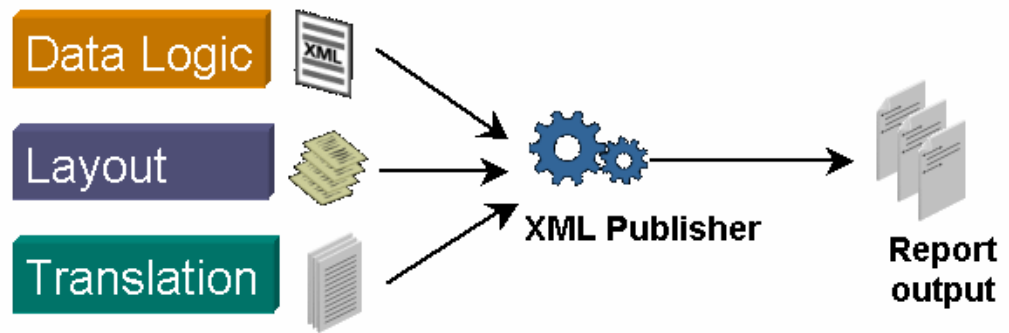
- Customer-ready PDF documents, such as financial statements, marketing materials, contracts, invoices, and purchase orders utilizing colors, images, font styles, headers and footers, and many other formatting and design options.
- HTML output for optimum online viewing.
- Excel output to create a spreadsheet of your report data.
- "Filled-out" third-party provided PDF documents. You can download a PDF document, such as a government form, to use as a template for your report. At runtime, the data and template produce a "filled-out" form.
- Flat text files to exchange with business partners for EDI and EFT transmission.

The following graphic displays a few sample documents generated by XML Publisher:



The flexibility is increased further by XML Publisher's ability to extract translatable strings from the report templates, allowing you to add translations to a report, independent of the layout and independent of the data generation.

With the approach of separating the data from the layout from the translation, it is possible to make changes in one of these layers without impacting the entire report package.



Moreover, because XML Publisher's design tools are integrated with well-known desktop applications, the report layouts can be created and modified by your business users, rather than having to rely on your technical staff.

## Template and Output Types

XML Publishers supports two basic template types: rich text format (RTF) and portable document file format (PDF).

**Note:** At runtime, XML Publisher converts RTF templates to XSL-FO. Therefore XSL-FO is also supported as a template type.

### RTF Templates

RTF templates can be created in Microsoft Word. Using a combination of native Microsoft Word features and XML Publisher command syntax you can create a report template that is ready to accept XML data from your system.

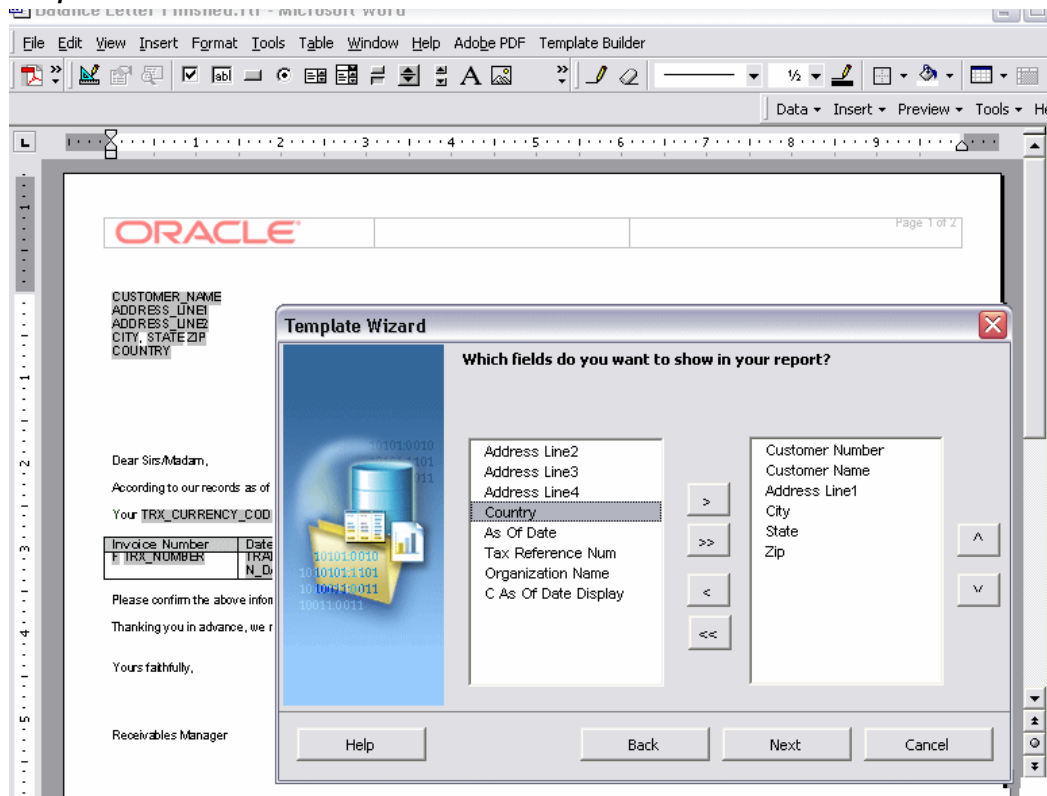
RTF templates can generate the following output types:

- PDF
- HTML
- RTF
- Excel
- XML

### RTF Design Tool

XML Publisher provides a plug-in for Microsoft Word to facilitate the design of RTF templates. The Business Intelligence Publisher Template Builder for Microsoft Word Add-in (formerly the XML Publisher Template Builder for Word) automates many tasks and provides preview capabilities. The desktop installation also includes samples and tutorials to get you started. The following figure shows the Template Builder for Word.

## Template Builder for Word



Creating an RTF Template, page 2-1 provides complete instructions for designing RTF templates.

### eText Templates

An eText template is a specialized type of RTF template used expressly for the generation of text output for Electronic Funds Transfer (EFT) and Electronic Data Interchange (EDI) transactions. To achieve the specialized layout, XML Publisher uses tables to define the position, length, and value of fields, as well as data manipulation commands.

At runtime XML Publisher constructs the output file according to the setup commands and layout specifications in the tables of the RTF template file.

Creating an eText Template, page 4-1 gives complete instructions for designing these table-based templates.

### PDF Templates

PDF templates are designed using Adobe Acrobat. You can use a PDF file from any source, including downloaded predefined forms (such as government forms). Using Acrobat's form fields, map your data source element names to the PDF fields where you want the data to appear.

PDF templates are less flexible than RTF templates, but are more appropriate for creating form-like reports, such as invoices or purchase orders.

See *Creating a PDF Template*, page 3-1 for detailed instructions.

## Translation Support

For RTF templates that you want available in different languages, XML Publisher provides the capability to extract the translatable strings from the template and export them to an industry-standard XLIFF file. The XLIFF file can then be translated in-house or shipped to a localization provider. You create one XLIFF file for each language and territory combination (locale) desired. The translated XLIFF files can then be loaded back to your system and associated with the original template file. At runtime, XML Publisher applies the template and translation appropriate for the user's selected locale.

See *Adding Template Translations*, page 6-1 for more information on translation support and working with XLIFF files.

## Configuring the Behavior of XML Publisher

XML Publisher's configuration file is a powerful tool for customizing the processing and output options for your system. Use this file to:

- Define a temporary directory to enable processing of large files and to enhance performance (setting this directory is **highly** recommended).
- Set security options for output PDFs, such as password protection and printing, copying, and modification permissions.
- Set options for each output type, such as change tracking for RTF output and viewing properties for HTML output.
- Define font mapping.
- Set options for XLIFF extraction, such as translation expansion percentage and minimum and maximum translation lengths.

See *Setting Runtime Properties*, page 7-1 for information on setting up the configuration file and the full list of properties.

## Using XML Publisher's Application Programming Interface

Developers who wish to create programs or applications that interact with XML Publisher can do so through its application programming interface. For more information see *Using the XML Publisher APIs*, page 8-1.

## Support for SQL, XSL, and XSL-FO

XML Publisher supports all native XSL commands in RTF templates and has extended a set of SQL functions for use with RTF templates. For information on using these commands in your templates, see *SQL, XSL, and XSL-FO Support*, page 5-1.

XML Publisher has not yet implemented the entire XSL-FO specification. For a complete list of the supported elements and attributes, see *Supported XSL-FO Elements*, page A-1.



---

## Creating an RTF Template

### Introduction

Rich Text Format (RTF) is a specification used by common word processing applications, such as Microsoft Word. When you save a document, RTF is a file type option that you select.

XML Publisher's RTF Template Parser converts documents saved as the RTF file type to XSL-FO. You can therefore create report designs using many of your standard word processing application's design features and XML Publisher will recognize and maintain the design.

During design time, you add data fields and other markup to your template using XML Publisher's simplified tags for XSL expressions. These tags associate the XML report data to your report layout. If you are familiar with XSL and prefer not to use the simplified tags, XML Publisher also supports the use of pure XSL elements in the template.

In addition to your word processing application's formatting features, XML Publisher supports other advanced reporting features such as conditional formatting, dynamic data columns, running totals, and charts.

If you wish to include code directly in your template, you can include *any* XSL element, many FO elements, and a set of SQL expressions extended by XML Publisher.

### Supported Modes

XML Publisher supports two methods for creating RTF templates:

- Basic RTF Method  
Use any word processing application that supports RTF version 1.6 writer (or later) to design a template using XML Publisher's simplified syntax.
- Form Field Method

Using Microsoft Word's form field feature allows you to place the syntax in hidden form fields, rather than directly into the design of your template. XML Publisher supports Microsoft Word 2000 (or later) with Microsoft Windows version 2000 (or later).

**Note:** If you use XSL or XSL:FO code rather than the simplified syntax, you must use the form field method.

This guide describes how to create RTF templates using both methods.

## Prerequisites

Before you design your template, you must:

- Know the business rules that apply to the data from your source report.
- Generate a sample of your source report in XML.
- Be familiar with the formatting features of your word processing application.

## Overview

Creating an RTF template file for use with XML Publisher consists of the following steps:

1. Generate sample data from your report.

You must have sample data either to reference while designing the report manually, or to load to the BI Publisher Template Builder for Word Add-in..

2. Load the data to the Template Builder for Word Add-in and use its features to add data fields, tables, charts, and other report items to your template.

Alternatively, insert the XML Publisher tags manually into your template, using the guidelines in this chapter.

3. Upload the template to the appropriate repository to make it available to XML Publisher at runtime.

When you design your template layout, you must understand how to associate the XML input file to the layout. This chapter presents a sample template layout with its input XML file to illustrate how to make the proper associations to add the markup tags to the template.

## Using the Business Intelligence Publisher Template Builder for Microsoft Word Add-in

The Template Builder is an extension to Microsoft Word that simplifies the

development of RTF templates. It automates many of the manual steps that are covered in this chapter. Use it in conjunction with this manual to increase your productivity.

**Note:** The BI Publisher Template Builder for Word Add-in includes features to log in to and interact with Oracle Business Intelligence Publisher Enterprise. These features only work with the Oracle BI Publisher Enterprise or Oracle BI Enterprise Edition implementations. See the Template Builder help for more information.

The Template Builder is tightly integrated with Microsoft Word and allows you to perform the following functions:

- Insert data fields
- Insert data-driven tables
- Insert data-driven forms
- Insert data-driven charts
- Preview your template with sample XML data
- Browse and update the content of form fields
- Extract boilerplate text into an XLIFF translation file and test translations

Manual steps for performing these functions are covered in this chapter. Instructions and tutorials for using the Template Builder are available from the readme and help files delivered with the tool.

## Associating the XML Data to the Template Layout

The following is a sample layout for a Payables Invoice Register:

### Sample Template Layout



### Payables Invoice Register

Page 1 of 1  
25<sup>th</sup> July 2003

Supplier:

Invoice Num	Invoice Date	GL Date	Curr	Entered Amt	Accounted Amt
Total for Supplier:					

Company Confidential

Note the following:

- The data fields that are defined on the template  
For example: Supplier, Invoice Number, and Invoice Date
- The elements of the template that will repeat when the report is run.  
For example, all the fields on the template will repeat for each Supplier that is reported. Each row of the invoice table will repeat for each invoice that is reported.

### XML Input File

Following is the XML file that will be used as input to the Payables Invoice Register report template:

**Note:** To simplify the example, the XML output shown below has been modified from the actual output from the Payables report.

```

<?xml version="1.0" encoding="WINDOWS-1252" ?>
- <VENDOR_REPORT>
- <LIST_G_VENDOR_NAME>
- <G_VENDOR_NAME>
  <VENDOR_NAME>COMPANY A</VENDOR_NAME>
- <LIST_G_INVOICE_NUM>
- <G_INVOICE_NUM>
  <SET_OF_BOOKS_ID>124</SET_OF_BOOKS_ID>
  <GL_DATE>10-NOV-03</GL_DATE>
  <INV_TYPE>Standard</INV_TYPE>
  <INVOICE_NUM>031110</INVOICE_NUM>
  <INVOICE_DATE>10-NOV-03</INVOICE_DATE>
  <INVOICE_CURRENCY_CODE>EUR</INVOICE_CURRENCY_CODE>
  <ENT_AMT>122</ENT_AMT>
  <ACCTD_AMT>122</ACCTD_AMT>
  <VAT_CODE>VAT22%</VAT_CODE>
</G_INVOICE_NUM>
</LIST_G_INVOICE_NUM>
<ENT_SUM_VENDOR>1000.00</ENT_SUM_VENDOR>
<ACCTD_SUM_VENDOR>1000.00</ACCTD_SUM_VENDOR>
</G_VENDOR_NAME>
</LIST_G_VENDOR_NAME>
<ACCTD_SUM_REP>108763.68</ACCTD_SUM_REP>
<ENT_SUM_REP>122039</ENT_SUM_REP>
</VENDOR_REPORT>

```

XML files are composed of elements. Each tag set is an element. For example `<INVOICE_DATE></INVOICE_DATE>` is the invoice date element. "INVOICE\_DATE" is the tag name. The data between the tags is the value of the element. For example, the value of INVOICE\_DATE is "10-NOV-03".

The elements of the XML file have a hierarchical structure. Another way of saying this is that the elements have parent-child relationships. In the XML sample, some elements are contained within the tags of another element. The containing element is the parent and the included elements are its children.

Every XML file has only one root element that contains all the other elements. In this example, VENDOR\_REPORT is the root element. The elements LIST\_G\_VENDOR\_NAME, ACCTD\_SUM\_REP, and ENT\_SUM\_REP are contained between the VENDOR\_REPORT tags and are children of VENDOR\_REPORT. Each child element can have child elements of its own.

## Identifying Placeholders and Groups

Your template content and layout must correspond to the content and hierarchy of the input XML file. Each data field in your template must map to an element in the XML file. Each group of repeating elements in your template must correspond to a parent-child relationship in the XML file.

To map the data fields you define *placeholders*. To designate the repeating elements, you define *groups*.

**Note:** XML Publisher supports regrouping of data if your report requires grouping that does not follow the hierarchy of your incoming

XML data. For information on using this feature, see Regrouping the XML Data, page 2-82.

## Placeholders

Each data field in your report template must correspond to an element in the XML file. When you mark up your template design, you define placeholders for the XML elements. The placeholder maps the template report field to the XML element. At runtime the placeholder is replaced by the value of the element of the same name in the XML data file.

For example, the "Supplier" field from the sample report layout corresponds to the XML element `VENDOR_NAME`. When you mark up your template, you create a placeholder for `VENDOR_NAME` in the position of the Supplier field. At runtime, this placeholder will be replaced by the value of the element from the XML file (the value in the sample file is **COMPANY A**).

## Identifying the Groups of Repeating Elements

The sample report lists suppliers and their invoices. There are fields that repeat for each supplier. One of these fields is the supplier's invoices. There are fields that repeat for each invoice. The report therefore consists of two groups of repeating fields:

- Fields that repeat for each supplier
- Fields that repeat for each invoice

The invoices group is nested inside the suppliers group. This can be represented as follows:

### Suppliers

- Supplier Name
- Invoices
  - Invoice Num
  - Invoice Date
  - GL Date
  - Currency
  - Entered Amount
  - Accounted Amount

- Total Entered Amount
- Total Accounted Amount

Compare this structure to the hierarchy of the XML input file. The fields that belong to the Suppliers group shown above are children of the element G\_VENDOR\_NAME. The fields that belong to the Invoices group are children of the element G\_INVOICE\_NUM.

By defining a group, you are notifying XML Publisher that *for each* occurrence of an element (parent), you want the included fields (children) displayed. At runtime, XML Publisher will loop through the occurrences of the element and display the fields each time.

## Designing the Template Layout

Use your word processing application's formatting features to create the design.

For example:

- Select the size, font, and alignment of text
- Insert bullets and numbering
- Draw borders around paragraphs
- Include a watermark
- Include images (jpg, gif, or png)
- Use table autoformatting features
- Insert a header and footer

For additional information on inserting headers and footers, see *Defining Headers and Footers*, page 2-15.

For a detailed list of supported formatting features in Microsoft Word, see *Supported Native Formatting Features*, page 2-41. Additional formatting and reporting features are described at the end of this section.

## Adding Markup to the Template Layout

XML Publisher converts the formatting that you apply in your word processing application to XSL-FO. You add markup to create the mapping between your layout and the XML file and to include features that cannot be represented directly in your format.

The most basic markup elements are placeholders, to define the XML data elements; and groups, to define the repeating elements.

XML Publisher provides tags to add markup to your template.

**Note:** For the XSL equivalents of the XML Publisher tags, see XSL Equivalent Syntax, page 5-6.

## Creating Placeholders

The placeholder maps the template field to the XML element data field. At runtime the placeholder is replaced by the value of the element of the same name in the XML data file.

Enter placeholders in your document using the following syntax:

```
<?XML element tag name?>
```

**Note:** The placeholder must match the XML element tag name exactly. It is case sensitive.

There are two ways to insert placeholders in your document:

1. Basic RTF Method: Insert the placeholder syntax directly into your template document.
2. Form Field Method: (Requires Microsoft Word) Insert the placeholder syntax in Microsoft Word's Text Form Field Options window. This method allows you to maintain the appearance of your template.

### Basic RTF Method

Enter the placeholder syntax in your document where you want the XML data value to appear.

Enter the element's XML tag name using the syntax:

```
<?XML element tag name?>
```

In the example, the template field "Supplier" maps to the XML element `VENDOR_NAME`. In your document, enter:

```
<?VENDOR_NAME?>
```

The entry in the template is shown in the following figure:



Supplier: <?VENDOR\_NAME?>

Invoice Num	Invoice

Total for Supplier:

## Form Field Method

Use Microsoft Word's **Text Form Field Options** window to insert the placeholder tags:

1. Enable the **Forms** toolbar in your Microsoft Word application.
2. Position your cursor in the place you want to create a placeholder.
3. Select the **Text Form Field** toolbar icon. This action inserts a form field area in your document.
4. Double-click the form field area to invoke the **Text Form Field Options** dialog box.
5. (Optional) Enter a description of the field in the **Default text** field. The entry in this field will populate the placeholder's position on the template.

For the example, enter "Supplier 1".

6. Select the **Add Help Text** button.
7. In the help text entry field, enter the XML element's tag name using the syntax:

`<?XML element tag name?>`

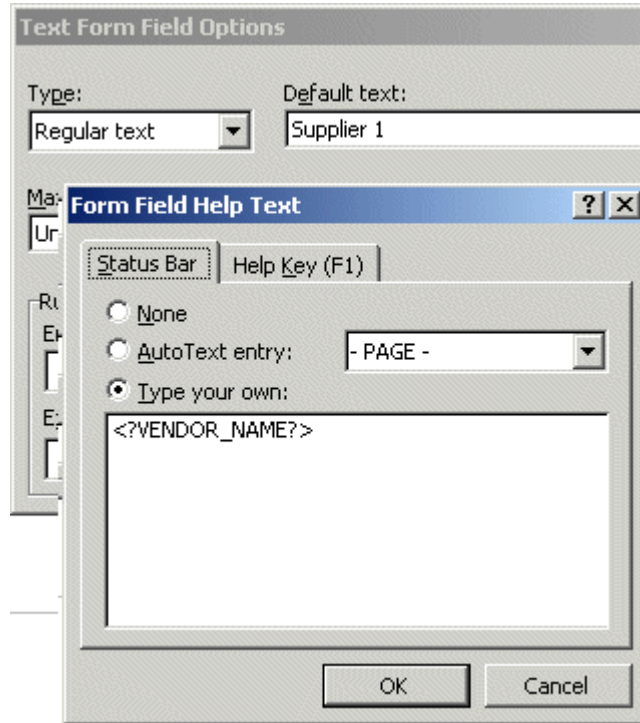
You can enter multiple element tag names in the text entry field.

In the example, the report field "Supplier" maps to the XML element VENDOR\_NAME. In the **Form Field Help Text** field enter:

`<?VENDOR_NAME?>`

The following figure shows the **Text Form Field Options** dialog box and the **Form Field Help Text** dialog box with the appropriate entries for the Supplier field.

**Tip:** For longer strings of XML Publisher syntax, use the Help Key (F1) tab instead of the Status Bar tab. The text entry field on the Help Key (F1) tab allows more characters.



8. Select **OK** to apply.

The **Default text** is displayed in the form field on your template.

The figure below shows the Supplier field from the template with the added form field markup.

+

Supplier: Supplier1

Invoice Num	Inv
Total	

## Complete the Example

The following table shows the entries made to complete the example. The Template

Field Name is the display name from the template. The Default Text Entry is the value entered in the Default Text field of the Text Form Field Options dialog box (form field method only). The Placeholder Entry is the XML element tag name entered either in the Form Field Help Text field (form field method) or directly on the template.

Template Field Name	Default Text Entry (Form Field Method)	Placeholder Entry (XML Tag Name)
Invoice Num	1234566	<?INVOICE_NUM?>
Invoice Date	1-Jan-2004	<?INVOICE_DATE?>
GL Date	1-Jan-2004	<?GL_DATE?>
Curr	USD	<?INVOICE_CURRENCY_CODE?>
Entered Amt	1000.00	<?ENT_AMT?>
Accounted Amt	1000.00	<?ACCTD_AMT?>
(Total of Entered Amt column)	1000.00	<?ENT_SUM_VENDOR?>
(Total of Accounted Amt column)	1000.00	<?ACCTD_SUM_VENDOR?>

The following figure shows the Payables Invoice Register with the completed form field placeholder markup.

See the Payables Invoice Register with Completed Basic RTF Markup, page 2-13 for the completed basic RTF markup.

Supplier: **Supplier 1**

Invoice Num	Invoice Date	GL Date	Curr	Entered Amt	Accounted Amt
1234566	1-Jan-2004	1-Jan-2004	USD	1000.00	1000.00
Total for Supplier: Supplier1				1000.00	1000.00

Company Confidential

## Defining Groups

By defining a group, you are notifying XML Publisher that *for each* occurrence of an element, you want the included fields displayed. At runtime, XML Publisher will loop through the occurrences of the element and display the fields each time.

In the example, for each occurrence of G\_VENDOR\_NAME in the XML file, we want the template to display its child elements VENDOR\_NAME (Supplier Name), G\_INVOICE\_NUM (the Invoices group), Total Entered Amount, and Total Accounted Amount. And, for each occurrence of G\_INVOICE\_NUM (Invoices group), we want the template to display Invoice Number, Invoice Date, GL Date, Currency, Entered Amount, and Accounted Amount.

To designate a group of repeating fields, insert the grouping tags around the elements to repeat.

Insert the following tag before the first element:

```
<?for-each:XML group element tag name?>
```

Insert the following tag after the final element:

```
<?end for-each?>
```

## Grouping scenarios

Note that the group element must be a parent of the repeating elements in the XML input file.

- If you insert the grouping tags around text or formatting elements, the text and formatting elements between the group tags will be repeated.

- If you insert the tags around a table, the table will be repeated.
- If you insert the tags around text in a table cell, the text in the table cell between the tags will be repeated.
- If you insert the tags around two different table cells, but in the same table row, the single row will be repeated.
- If you insert the tags around two different table rows, the rows between the tags will be repeated (this does not include the row that contains the "end group" tag).

## Basic RTF Method

Enter the tags in your document to define the beginning and end of the repeating element group.

To create the Suppliers group in the example, insert the tag

```
<?for-each:G_VENDOR_NAME?>
```

before the Supplier field that you previously created.

Insert `<?end for-each?>` in the document after the summary row.

The following figure shows the Payables Invoice Register with the basic RTF grouping and placeholder markup:



### Payables Invoice Register

Page 1 of 1  
25<sup>th</sup> January 2004

```
<?for-each:G_VENDOR_NAME?>    <?sort:VENDOR_NAME?>
```

Supplier: <?VENDOR\_NAME?>

Invoice Num	Invoice Date	GL Date	Curr	Entered Amt	Accounted Amt
<?for-each: G_INVOICE_NUM?> <?INVOICE_NUM?>	<?INVOICE_D ATE?>	<?GL_DATE?>	<?INV OICE CUR REN CY_C ODE?>	<?ENT_AMT?>	<?ACCTD_AMT?> <?end for-each?>

Total for Supplier: <?VENDOR_NAME?>	<?ENT_SUM_VEND OR?>	<?ACCTD_SUM_V ENDOR?>
-------------------------------------	------------------------	--------------------------

<?end for-each?>

Company Confidential

## Form Field Method

1. Insert a form field to designate the beginning of the group.

In the help text field enter:

<?for-each:group element tag name?>

To create the Suppliers group in the example, insert a form field before the Suppliers field that you previously created. In the help text field enter:

<?for-each:G\_VENDOR\_NAME?>

For the example, enter the Default text "Group: Suppliers" to designate the beginning of the group on the template. The Default text is not required, but can make the template easier to read.

2. Insert a form field after the final placeholder element in the group. In the help text field enter <?end for-each?>.

For the example, enter the Default text "End: Suppliers" after the summary row to designate the end of the group on the template.

The following figure shows the template after the markup to designate the Suppliers group was added.

Group: Suppliers

Supplier: Supplier 1

Invoice Num	Invoice
1234566	1-Jan-

End:Suppliers

## Complete the Example

The second group in the example is the invoices group. The repeating elements in this group are displayed in the table. For each invoice, the table row should repeat. Create a group within the table to contain these elements.

**Note:** For each invoice, only the table *row* should repeat, not the entire table. Placing the grouping tags at the beginning and end of the table row will repeat only the row. If you place the tags around the table, then for each new invoice the entire table with headings will be

repeated.

To mark up the example, insert the grouping tag `<?for-each:G_INVOICE_NUM?>` in the table cell before the Invoice Num placeholder. Enter the Default text "Group:Invoices" to designate the beginning of the group.

Insert the end tag inside the final table cell of the row after the Accounted Amt placeholder. Enter the Default text "End:Invoices" to designate the end of the group.

The following figure shows the completed example using the form field method:



## Payables Invoice Register

Page 1 of 1  
25<sup>th</sup> July 2003

Group: Suppliers Sort by: Supplier

Supplier: **Supplier 1**

Invoice Num	Invoice Date	GL Date	Curr	Entered Amt	Accounted Amt
Group:Invoices 1234566	1-Jan-2004	1-Jan-2004	USD	1000.00	1000.00
Total for Supplier: Supplier1				1000.00	1000.00

End:Suppliers

Company Confidential

## Defining Headers and Footers

### Native Support

XML Publisher supports the use of the native RTF header and footer feature. To create a header or footer, use the your word processing application's header and footer insertion tools. As an alternative, or if you have multiple headers and footers, you can use `start:body` and `end body` tags to distinguish the header and footer regions from the body of your report.

### Inserting Placeholders in the Header and Footer

At the time of this writing, Microsoft Word does not support form fields in the header and footer. You must therefore insert the placeholder syntax directly into the template (basic RTF method), or use the start body/end body syntax described in the next section.

## Multiple or Complex Headers and Footers

If your template requires multiple headers and footers, create them by using XML Publisher tags to define the body area of your report. You may also want to use this method if your header and footer contain complex objects that you wish to place in form fields. When you define the body area, the elements occurring before the beginning of the body area will compose the header. The elements occurring after the body area will compose the footer.

Use the following tags to enclose the body area of your report:

```
<?start:body?>
```


```
<?end body?>
```

Use the tags either directly in the template, or in form fields.

The Payables Invoice Register contains a simple header and footer and therefore does not require the start body/end body tags. However, if you wanted to add another header to the template, define the body area as follows:

1. Insert `<?start:body?>` before the Suppliers group tag:  
`<?for-each:G_VENDOR_NAME?>`
2. Insert `<?end body?>` after the Suppliers group closing tag: `<?end for-each?>`

The following figure shows the Payables Invoice Register with the start body/end body tags inserted:



**Payables Invoice Register**

Page 1 of 1  
25<sup>th</sup> July 2003

<?start:body?>

Group: Suppliers    Sort by Supplier

Supplier: **Supplier 1**

Invoice Num	Invoice Date	GL Date	Curr	Entered Amt	Accounted Amt
Group:Invoices 1234566	1-Jan-2004	1-Jan-2004	USD	1000.00	1000.00
				End:Invoices	
Total for Supplier: Supplier1				1000.00	1000.00

End:Suppliers  
<?end body?>

Company Confidential

## Different First Page and Different Odd and Even Page Support

If your report requires a different header and footer on the first page of your report; or, if your report requires different headers and footers for odd and even pages, you can define this behavior using Microsoft Word's Page Setup dialog.

1. Select **Page Setup** from the **File** menu.



2. In the **Page Setup** dialog, select the **Layout** tab.
3. In the **Headers and footers** region of the dialog, select the appropriate check box:  
Different odd and even  
Different first page
4. Insert your headers and footers into your template as desired.

At runtime your generated report will exhibit the defined header and footer behavior.

## Images and Charts

### Images

XML Publisher supports several methods for including images in your published document:

#### Direct Insertion

Insert the jpg, gif, bmp, or png image directly in your template.

#### URL Reference

URL Reference

1. Insert a dummy image in your template.
2. In Microsoft Word's **Format Picture** dialog box select the **Web** tab. Enter the following syntax in the **Alternative text** region to reference the image URL:

```
url: {'http://image location'}
```

For example, enter:

```
url: {'http://www.oracle.com/images/ora_log.gif'}
```

#### OA Media Directory Reference

**Note:** This method only applies to Oracle E-Business Suite installations.

1. Insert a dummy image in your template.
2. In Microsoft Word's **Format Picture** dialog box select the **Web** tab. Enter the following syntax in the **Alternative text** region to reference the OA\_MEDIA directory:

```
url: {'${OA_MEDIA}/image name'}
```

For example, enter:

```
url: {'${OA_MEDIA}/ORACLE_LOGO.gif'}
```

### Element Reference from XML File

1. Insert a dummy image in your template.
2. In Microsoft Word's **Format Picture** dialog box select the **Web** tab. Enter the following syntax in the **Alternative text** region to reference the image URL:

```
url:{IMAGE_LOCATION}
```

where IMAGE\_LOCATION is an element from your XML file that holds the full URL to the image.

You can also build a URL based on multiple elements at runtime. Just use the `concat` function to build the URL string. For example:

```
url:{concat(SERVER, '/', IMAGE_DIR, '/', IMAGE_FILE) }
```

where SERVER, IMAGE\_DIR, and IMAGE\_FILE are element names from your XML file that hold the values to construct the URL.

This method can also be used with the OA\_MEDIA reference as follows:

```
url:{concat('${OA_MEDIA}', '/', IMAGE_FILE) }
```

### Rendering an Image Retrieved from BLOB Data

**Important:** This section applies to Oracle E-Business Suite and Oracle Business Intelligence implementations only.

If your data source is a Data Template and your results XML contains image data that had been stored as a BLOB in the database, use the following syntax in a form field inserted in your template where you want the image to render at runtime:

```
<fo:instream-foreign-object content type="image/jpg">  
<xsl:value-of select="IMAGE_ELEMENT"/>  
</fo:instream-foreign-object>
```

where

*image/jpg* is the MIME type of the image (other options might be: *image/gif* and *image/png*)

and

*IMAGE\_ELEMENT* is the element name of the BLOB in your XML data.

Note that you can specify `height` and `width` attributes for the image to set its size in the published report. XML Publisher will scale the image to fit the box size that you define. For example, to set the size of the example above to three inches by four inches, enter the following:

```
<fo:instream-foreign-object content type="image/jpg" height="3 in"
width="4 in">
<xsl:value-of select="IMAGE_ELEMENT"/>
</fo:instream-foreign-object>
```

Specify in pixels as follows:

```
<fo:instream-foreign-object content type="image/jpg" height="300 px"
width="4 px">
...
```

or in centimeters:

```
<fo:instream-foreign-object content type="image/jpg" height="3 cm"
width="4 cm">
...
```

or as a percentage of the original dimensions:

```
<fo:instream-foreign-object content type="image/jpg" height="300%"
width="300%">
...
```

## Chart Support

XML Publisher leverages the graph capabilities of Oracle Business Intelligence Beans (BI Beans) to enable you to define charts and graphs in your RTF templates that will be populated with data at runtime. XML Publisher supports all the graph types and component attributes available from the BI Beans graph DTD.

The BI Beans graph DTD is fully documented in the following technical note available from the Oracle Technology Network [<http://www.oracle.com/technology/index.html>] (OTN): "DTD for Customizing Graphs in Oracle Reports [[http://www.oracle.com/technology/products/reports/htdocs/getstart/whitepapers/graph\\_hdttd/graph\\_dtd\\_technote\\_2.html](http://www.oracle.com/technology/products/reports/htdocs/getstart/whitepapers/graph_hdttd/graph_dtd_technote_2.html) ]."

The following summarizes the steps to add a chart to your template. These steps will be discussed in detail in the example that follows:

1. Insert a dummy image in your template to define the size and position of your chart.
2. Add the definition for the chart to the Alternative text box of the dummy image. The chart definition requires XSL commands.
3. At runtime XML Publisher calls the BI Beans applications to render the image that is then inserted into the final output document.

## Adding a Sample Chart

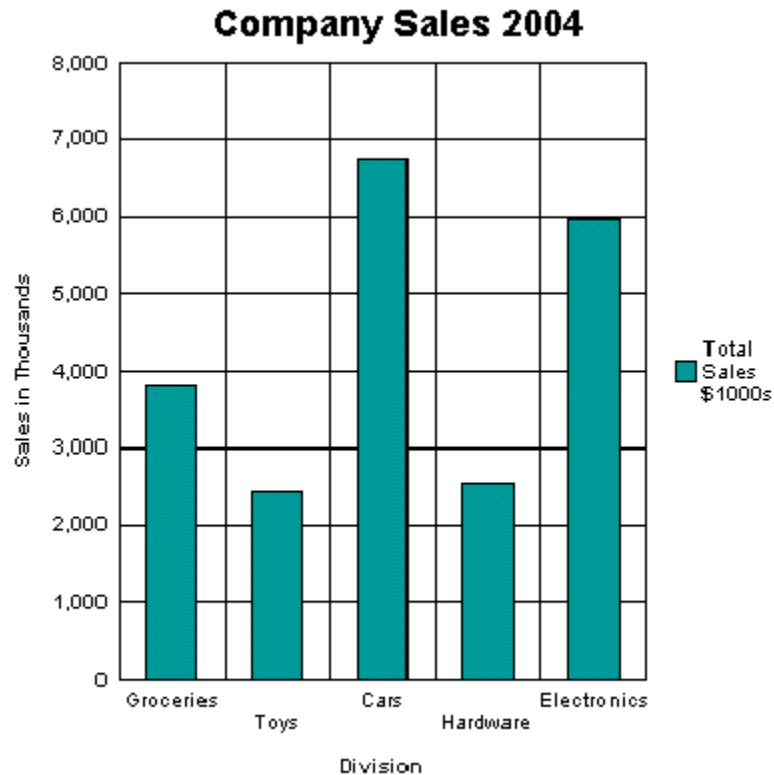
Following is a piece of XML data showing total sales by company division.

```

<sales year=2004>
  <division>
    <name>Groceries</name>
    <totalsales>3810</totalsales>
    <costofsales>2100</costofsales>
  </division>
  <division>
    <name>Toys</name>
    <totalsales>2432</totalsales>
    <costofsales>1200</costofsales>
  </division>
  <division>
    <name>Cars</name>
    <totalsales>6753</totalsales>
    <costofsales>4100</costofsales>
  </division>
  <division>
    <name>Hardware</name>
    <totalsales>2543</totalsales>
    <costofsales>1400</costofsales>
  </division>
  <division>
    <name>Electronics</name>
    <totalsales>5965</totalsales>
    <costofsales>3560</costofsales>
  </division>
</sales>

```

This example will show how to insert a chart into your template to display it as a vertical bar chart as shown in the following figure:



Note the following attributes of this chart:

- The style is a vertical bar chart.
- The chart displays a background grid.
- The components are colored.
- Sales totals are shown as Y-axis labels.
- Divisions are shown as X-axis labels.
- The chart is titled.
- The chart displays a legend.

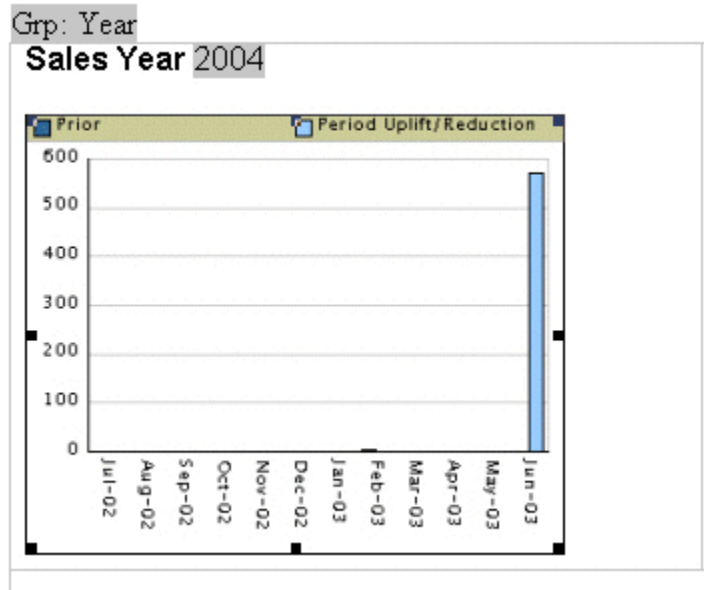
Each of these properties can be customized to suit individual report requirements.

#### Inserting the Dummy Image

The first step is to add a dummy image to the template in the position you want the chart to appear. The image size will define how big the chart image will be in the final document.

**Important:** You must insert the dummy image as a "Picture" and not any other kind of object.

The following figure shows an example of a dummy image:

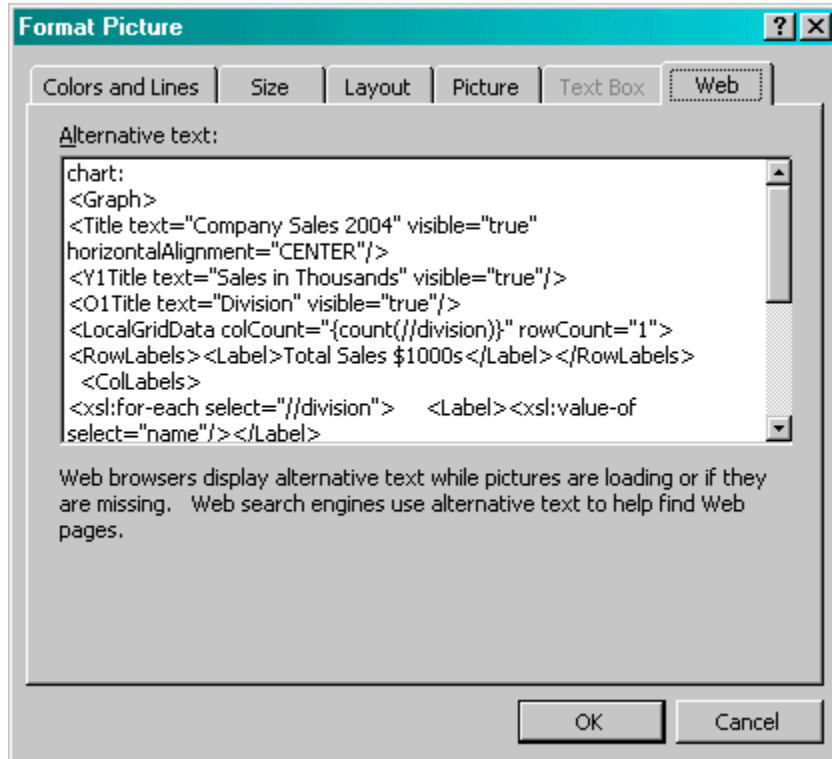


The image can be embedded inside a for-each loop like any other form field if you want the chart to be repeated in the output based on the repeating data. In this example, the chart is defined within the sales year group so that a chart will be generated for each year of data present in the XML file.

Right-click the image to open the **Format Picture** palette and select the **Web** tab. Use the **Alternative text** entry box to enter the code to define the chart characteristics and data definition for the chart.

#### **Adding Code to the Alternative Text Box**

The following graphic shows an example of the XML Publisher code in the **Format Picture Alternative text** box:



The content of the **Alternative text** represents the chart that will be rendered in the final document. For this chart, the text is as follows:

```

chart:
<Graph graphType = "BAR_VERT_CLUST">
  <Title text="Company Sales 2004" visible="true"
horizontalAlignment="CENTER"/>
  <Y1Title text="Sales in Thousands" visible="true"/>
  <OI1Title text="Division" visible="true"/>
  <LocalGridData colCount="{count(//division)}" rowCount="1">
    <RowLabels>
      <Label>Total Sales $1000s</Label>
    </RowLabels>
    <ColLabels>
      <xsl:for-each select="//division">
        <Label>
          <xsl:value-of select="name"/>
        </Label>
      </xsl:for-each>
    </ColLabels>
    <DataValues>
      <RowData>
        <xsl:for-each select="//division">
          <Cell>
            <xsl:value-of select="totalsales"/>
          </Cell>
        </xsl:for-each>
      </RowData>
    </DataValues>
  </LocalGridData>
</Graph>

```

The first element of your chart text must be the `chart:` element to inform the RTF parser that the following code describes a chart object.

Next is the opening `<Graph>` tag. Note that the whole of the code resides within the tags of the `<Graph>` element. This element has an attribute to define the chart type: `graphType`. If this attribute is not declared, the default chart is a vertical bar chart. BI Beans supports many different chart types. Several more types are presented in this section. For a complete listing, see the BI Beans graph DTD documentation.

The following code section defines the chart type and attributes:

```
<Title text="Company Sales 2004" visible="true"
horizontalAlignment="CENTER"/>
  <YlTitle text="Sales in Thousands" visible="true"/>
  <OlTitle text="Division" visible="true"/>
```

All of these values can be declared or you can substitute values from the XML data at runtime. For example, you can retrieve the chart title from an XML tag by using the following syntax:

```
<Title text="{CHARTTITLE}" visible="true" horizontalAlignment="CENTER"/>
```

where "CHARTTITLE" is the XML tag name that contains the chart title. Note that the tag name is enclosed in curly braces.

The next section defines the column and row labels:

```
<LocalGridData colCount="{count(//division)}" rowCount="1">
  <RowLabels>
    <Label>Total Sales $1000s</Label>
  </RowLabels>
  <ColLabels>
    <xsl:for-each select="//division">
      <Label>
        <xsl:value-of select="name"/>
      </Label>
    </xsl:for-each>
  </ColLabels>
```

The `LocalGridData` element has two attributes: `colCount` and `rowCount`. These define the number of columns and rows that will be shown at runtime. In this example, a count function calculates the number of columns to render:

```
colCount="{count(//division)}"
```

The `rowCount` has been hard-coded to 1. This value defines the number of sets of data to be charted. In this case it is 1.

Next the code defines the row and column labels. These can be declared, or a value from the XML data can be substituted at runtime. The row label will be used in the chart legend (that is, "Total Sales \$1000s").

The column labels for this example are derived from the data: Groceries, Toys, Cars, and so on. This is done using a for-each loop:



```

<ColLabels>
  <xsl:for-each select="//division">
    <Label>
      <xsl:value-of select="name"/>
    </Label>
  </xsl:for-each>
</ColLabels>

```

This code loops through the <division> group and inserts the value of the <name> element into the <Label> tag. At runtime, this will generate the following XML:

```

<ColLabels>
  <Label>Groceries</Label>
  <Label>Toys</Label>
  <Label>Cars</Label>
  <Label>Hardware</Label>
  <Label>Electronics</Label>
</ColLabels>

```

The next section defines the actual data values to chart:

```

<DataValues>
  <RowData>
    <xsl:for-each select="//division">
      <Cell>
        <xsl:value-of select="totalsales"/>
      </Cell>
    </xsl:for-each>
  </RowData>
</DataValues>

```

Similar to the labels section, the code loops through the data to build the XML that is passed to the BI Beans rendering engine. This will generate the following XML:

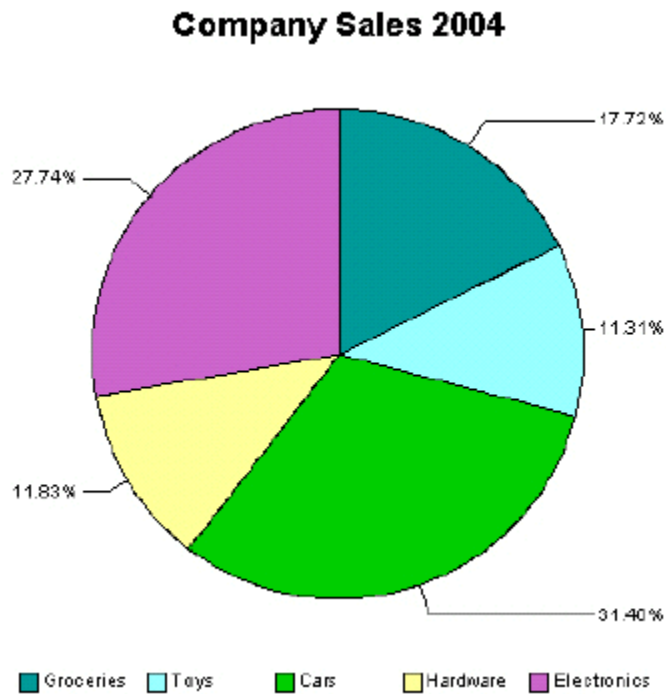
```

<DataValues>
  <RowData>
    <Cell>3810</Cell>
    <Cell>2432</Cell>
    <Cell>6753</Cell>
    <Cell>2543</Cell>
    <Cell>5965</Cell>
  </RowData>
</DataValues>

```

## Additional Chart Samples

You can also display this data in a pie chart as shown in the following figure:

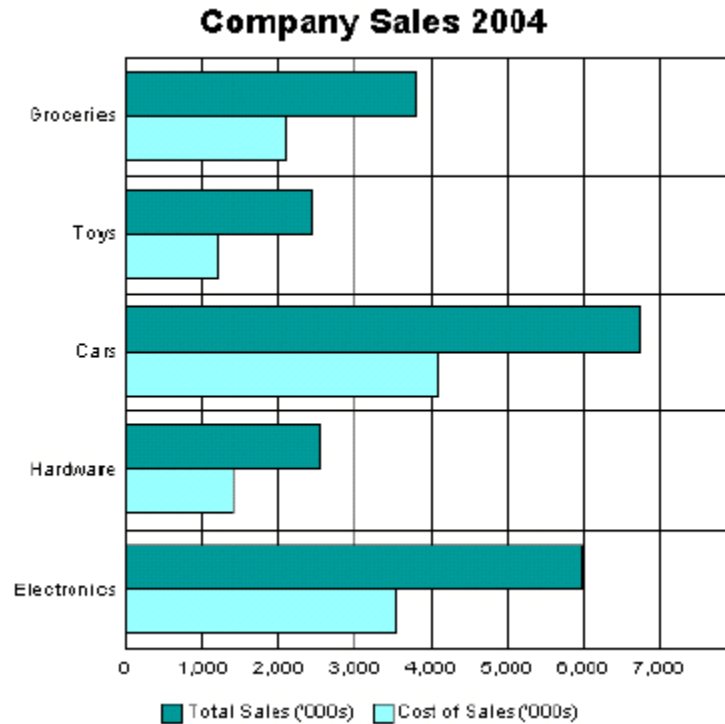


The following is the code added to the template to render this chart at runtime:

```
chart:
<Graph graphType="PIE">
  <Title text="Company Sales 2004" visible="true"
    horizontalAlignment="CENTER"/>
  <LocalGridData rowCount="{count(//division)}" colCount="1">
    <RowLabels>
      <xsl:for-each select="//division">
        <Label>
          <xsl:value-of select="name"/>
        </Label>
      </xsl:for-each>
    </RowLabels>
    <DataValues>
      <xsl:for-each select="//division">
        <RowData>
          <Cell>
            <xsl:value-of select="totalsales"/>
          </Cell>
        </RowData>
      </xsl:for-each>
    </DataValues>
  </LocalGridData>
</Graph>
```

### Horizontal Bar Chart Sample

The following example shows total sales and cost of sales charted in a horizontal bar format. This example also adds the data from the cost of sales element (`<costofsales>`) to the chart:



The following code defines this chart in the template:

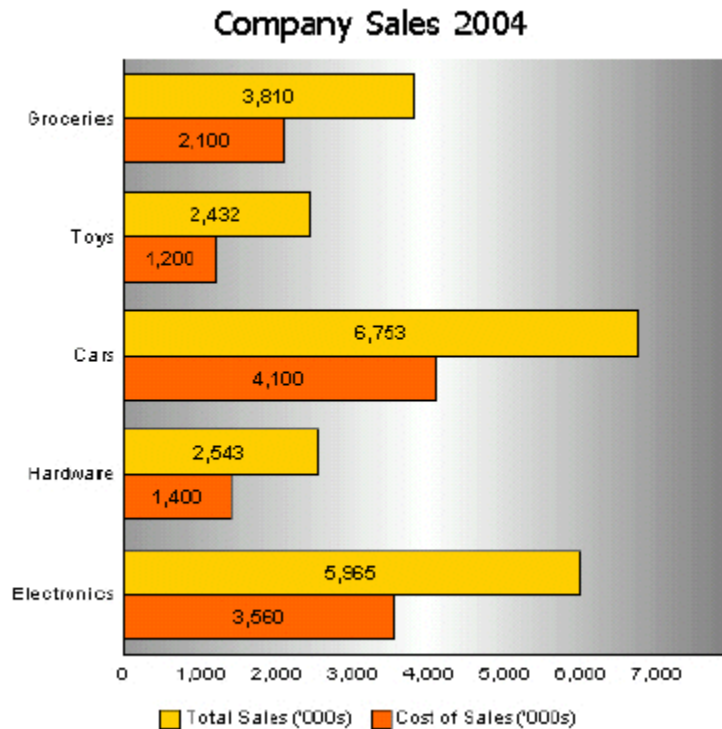
```
chart:
<Graph graphType = "BAR_HORIZ_CLUST">
  <Title text="Company Sales 2004" visible="true"
horizontalAlignment="CENTER"/>
  <LocalGridData colCount="{count(//division)}" rowCount="2">
    <RowLabels>
      <Label>Total Sales ('000s)</Label>
      <Label>Cost of Sales ('000s)</Label>
    </RowLabels>
    <ColLabels>
      <xsl:for-each select="//division">
        <Label><xsl:value-of select="name"/></Label>
      </xsl:for-each>
    </ColLabels>
    <DataValues>
      <RowData>
        <xsl:for-each select="//division">
          <Cell><xsl:value-of select="totalsales"/></Cell>
        </xsl:for-each>
      </RowData>
      <RowData>
        <xsl:for-each select="//division">
          <Cell><xsl:value-of select="costofsales"/></Cell>
        </xsl:for-each>
      </RowData>
    </DataValues>
  </LocalGridData>
</Graph>
```

To accommodate the second set of data, the `rowCount` attribute for the

LocalGridData element is set to 2. Also note the DataValues section defines two sets of data: one for Total Sales and one for Cost of Sales.

## Changing the Appearance of Your Chart

There are many attributes available from the BI Beans graph DTD that you can manipulate to change the look and feel of your chart. For example, the previous chart can be changed to remove the grid, place a graduated background, and change the bar colors and fonts as shown in the following figure:



The code to support this is as follows:

```

chart:
<Graph graphType = "BAR_HORIZ_CLUST">
<SeriesItems>
  <Series id="0" color="#ffcc00"/>
  <Series id="1" color="#ff6600"/>
</SeriesItems>
<O1MajorTick visible="false"/>
<X1MajorTick visible="false"/>
<Y1MajorTick visible="false"/>
<Y2MajorTick visible="false"/>
<MarkerText visible="true" markerTextPlace="MTP_CENTER"/>
<PlotArea borderTransparent="true">
  <SFX fillType="FT_GRADIENT" gradientDirection="GD_LEFT"
    gradientNumPins="300">
    <GradientPinStyle pinIndex="1" position="1"
      gradientPinLeftColor="#999999"
      gradientPinRightColor="#cc6600"/>
  </SFX>
</PlotArea>
<Title text="Company Sales 2004" visible="true">
  <GraphFont name="Tahoma" bold="false"/>
</Title>
. . .
</Graph>

```

The colors for the bars are defined in the `SeriesItems` section. The colors are defined in hexadecimal format as follows:

```

<SeriesItems>
  <Series id="0" color="#ffcc00"/>
  <Series id="1" color="#ff6600"/>
</SeriesItems>

```

The following code hides the chart grid:

```

<O1MajorTick visible="false"/>
<X1MajorTick visible="false"/>
<Y1MajorTick visible="false"/>
<Y2MajorTick visible="false"/>

```

The `MarkerText` tag places the data values on the chart bars:

```

<MarkerText visible="true" markerTextPlace="MTP_CENTER"/>

```

The `PlotArea` section defines the background. The `SFX` element establishes the gradient and the `borderTransparent` attribute hides the plot border:

```

<PlotArea borderTransparent="true">
  <SFX fillType="FT_GRADIENT" gradientDirection="GD_LEFT"
    gradientNumPins="300">
    <GradientPinStyle pinIndex="1" position="1"
      gradientPinLeftColor="#999999"
      gradientPinRightColor="#cc6600"/>
  </SFX>
</PlotArea>

```

The `Title text` tag has also been updated to specify a new font type and size:

```

<Title text="Company Sales 2004" visible="true">
  <GraphFont name="Tahoma" bold="false"/>
</Title>

```

## Drawing, Shape and Clip Art Support

XML Publisher supports Microsoft Word drawing, shape, and clip art features. You can add these objects to your template and they will be rendered in your final PDF output.

The following AutoShape categories are supported:

- Lines - straight, arrowed, connectors, curve, free form, and scribble
- Connectors - straight connectors only are supported. Curved connectors can be achieved by using a curved line and specifying the end styles to the line.
- Basic Shapes - all shapes are supported.
- Block arrows - all arrows are supported.
- Flowchart - all flowchart objects are supported.
- Stars and Banners - all objects are supported.
- Callouts - the "line" callouts are not supported.
- Clip Art - add images to your templates using the Microsoft Clip Art libraries

### Freehand Drawing

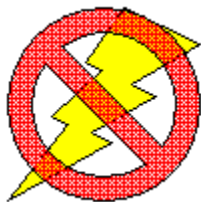
Use the freehand drawing tool in Microsoft Word to create drawings in your template to be rendered in the final PDF output.

### Hyperlinks

You can add hyperlinks to your shapes. See [Hyperlinks](#), page 2-54.

### Layering

You can layer shapes on top of each other and use the transparency setting in Microsoft Word to allow shapes on lower layers to show through. The following graphic shows an example of layered shapes:



### 3-D Effects

XML Publisher does not currently support the 3-D option for shapes.

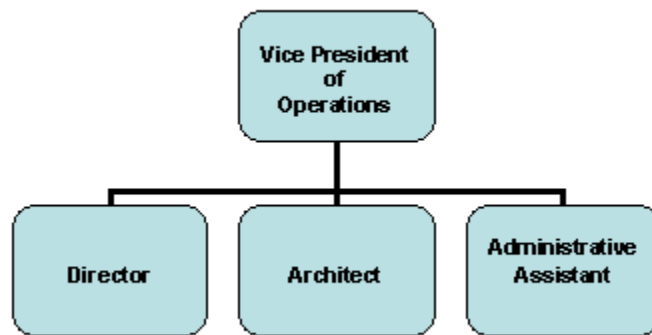
### Microsoft Equation

Use the equation editor to generate equations in your output. The following figure shows an example of an equation:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N \left( x_i - \bar{x} \right)^2}$$

### Organization Chart

Use the organization chart functionality in your templates and the chart will be rendered in the output. The following image shows an example of an organization chart:



### WordArt

You can use Microsoft Word's WordArt functionality in your templates. The following graphic shows a WordArt example:



**Note:** Some Microsoft WordArt uses a bitmap operation that currently cannot be converted to SVG. To use the unsupported WordArt in your template, you can take a screenshot of the WordArt then save it as an image (gif, jpeg, or png) and replace the WordArt with the image.

## Data Driven Shape Support

In addition to supporting the static shapes and features in your templates, XML Publisher supports the manipulation of shapes based on incoming data or parameters, as well. The following manipulations are supported:

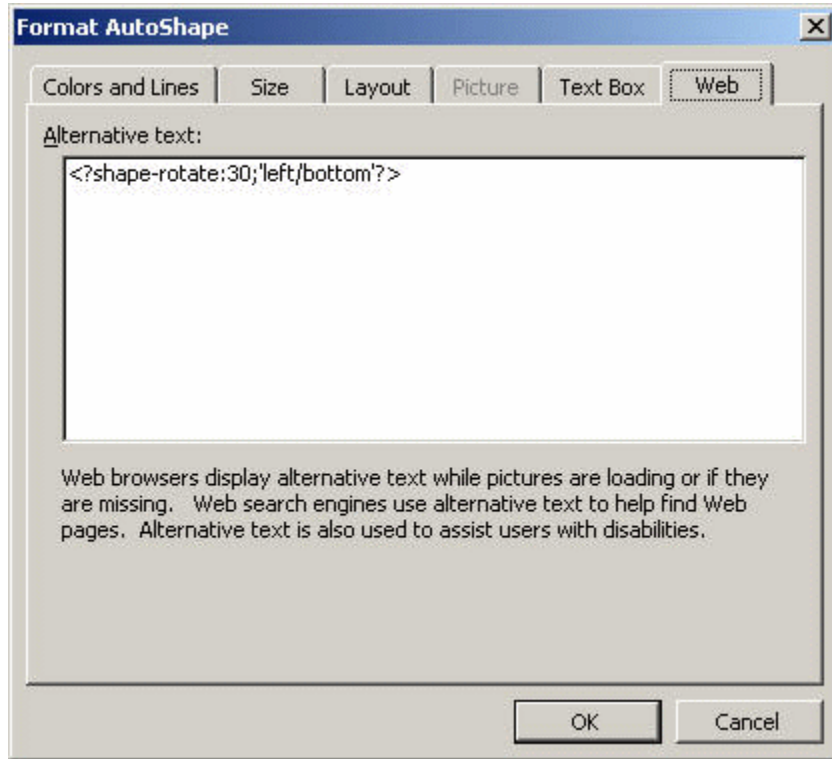
- Replicate
- Move
- Change size
- Add text
- Skew
- Rotate

These manipulations not only apply to single shapes, but you can use the group feature in Microsoft Word to combine shapes together and manipulate them as a group.

## Placement of Commands

Enter manipulation commands for a shape in the Web tab of the shape's properties dialog as shown in the following example figure:





## Replicate a Shape

You can replicate a shape based on incoming XML data in the same way you replicate data elements in a for-each loop. To do this, use a for-each@shape command in conjunction with a shape-offset declaration. For example, to replicate a shape down the page, use the following syntax:

```
<?for-each@shape:SHAPE_GROUP?>
  <?shape-offset-y: (position()-1)*100?>
<?end for-each?>
```

where

for-each@shape opens the for-each loop for the shape context

SHAPE\_GROUP is the name of the repeating element from the XML file. For each occurrence of the element SHAPE\_GROUP a new shape will be created.

shape-offset-y: - is the command to offset the shape along the y-axis.

(position()-1)\*100) - sets the offset in pixels per occurrence. The XSL position command returns the record counter in the group (that is 1,2,3,4); one is subtracted from that number and the result is multiplied by 100. Therefore for the first occurrence the offset would be 0: (1-1) \* 100. The offset for the second occurrence would be 100 pixels: (2-1) \*100. And for each subsequent occurrence the offset would be another 100 pixels down the page.

### Add Text to a Shape

You can add text to a shape dynamically either from the incoming XML data or from a parameter value. In the property dialog enter the following syntax:

```
<?shape-text:SHAPETEXT?>
```

where SHAPETEXT is the element name in the XML data. At runtime the text will be inserted into the shape.

### Add Text Along a Path

You can add text along a line or curve from incoming XML data or a parameter. After drawing the line, in the property dialog enter:

```
<?shape-text-along-path:SHAPETEXT?>
```

where SHAPETEXT is the element from the XML data. At runtime the value of the element SHAPETEXT will be inserted above and along the line.

### Moving a Shape

You can move a shape or transpose it along both the x and y-axes based on the XML data. For example to move a shape 200 pixels along the y-axis and 300 along the x-axis, enter the following commands in the property dialog of the shape:

```
<?shape-offset-x:300?>
```

```
<?shape-offset-y:200?>
```

### Rotating a Shape

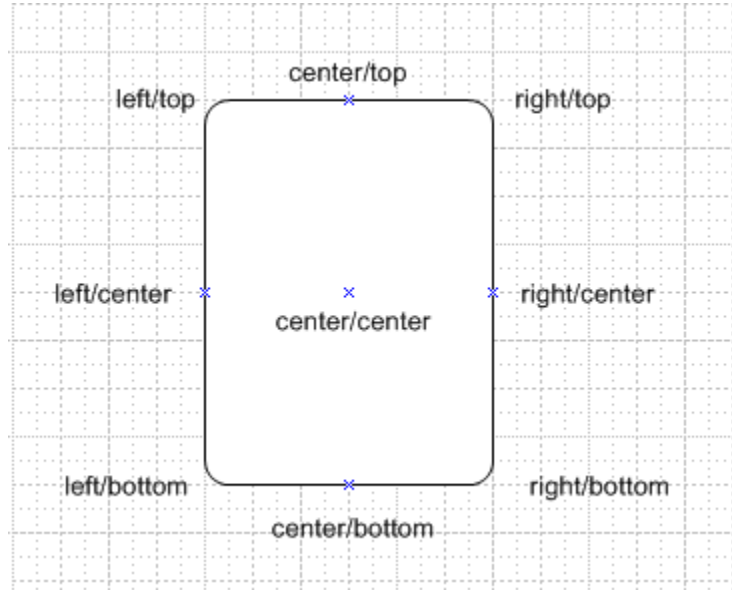
To rotate a shape about a specified axis based on the incoming data, use the following command:

```
<?shape-rotate:ANGLE;'POSITION'?>
```

where

ANGLE is the number of degrees to rotate the shape. If the angle is positive, the rotation is clockwise; if negative, the rotation is counterclockwise.

POSITION is the point about which to carry out the rotation, for example, 'left/top'. Valid values are combinations of left, right, or center with center, top, or bottom. The default is left/top. The following figure shows these valid values:



To rotate this rectangle shape about the bottom right corner, enter the following syntax:

```
<?shape-rotate:60,'right/bottom'?>
```

You can also specify an x,y coordinate within the shape itself about which to rotate.

### Skewing a Shape

You can skew a shape along its x or y axis using the following commands:

```
<?shape-skew-x:ANGLE;' POSITION'?>
<?shape-skew-y:ANGLE;' POSITION'?>
```

where

ANGLE is the number of degrees to skew the shape. If the angle is positive, the skew is to the right.

POSITION is the point about which to carry out the rotation, for example, 'left/top'. Valid values are combinations of left, right, or center with center, top, or bottom. See the figure under Rotating a Shape, page 2-34. The default is 'left/top'.

For example, to skew a shape by 30 degrees about the bottom right hand corner, enter the following:

```
<?shape-skew-x:number(.)*30;'right/bottom'?>
```

### Changing the Size of a Shape

You can change the size of a shape using the appropriate commands either along a single axis or both axes. To change a shape's size along both axes, use:

```
<?shape-size:RATIO?>
```

where RATIO is the numeric ratio to increase or decrease the size of the shape. Therefore a value of 2 would generate a shape twice the height and width of the

original. A value of 0.5 would generate a shape half the size of the original.

To change a shape's size along the x or y axis, use:

```
<?shape-size-x:RATIO?>  
<?shape-size-y:RATIO?>
```

Changing only the x or y value has the effect of stretching or shrinking the shape along an axis. This can be data driven.

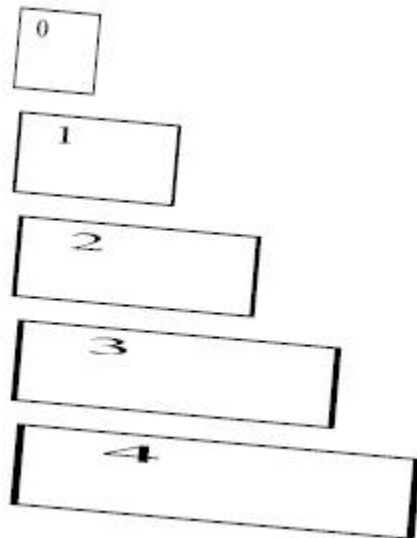
### Combining Commands

You can also combine these commands to carry out multiple transformations on a shape at one time. For example, you can replicate a shape and for each replication, rotate it by some angle and change the size at the same time.

The following example shows how to replicate a shape, move it 50 pixels down the page, rotate it by five degrees about the center, stretch it along the x-axis and add the number of the shape as text:

```
<for-each@shape:SHAPE_GROUP?>  
  <?shape-text:position()?>  
  <?shape-offset-y:position()*50?>  
  <?shape-rotate:5;'center/center'?>  
  <?shape-size-x:position()+1?>  
<end for-each?>
```

This would generate the output shown in the following figure:



### CD Ratings Example

This example demonstrates how to set up a template that will generate a star-rating based on data from an incoming XML file.

Assume the following incoming XML data:

```

<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
    <USER_RATING>4</USER_RATING>
  </CD>
  <CD>
    <TITLE>Hide Your Heart</TITLE>
    <ARTIST>Bonnie Tylor</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
    <USER_RATING>3</USER_RATING>
  </CD>
  <CD>
    <TITLE>Still got the blues</TITLE>
    <ARTIST>Gary More</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>Virgin Records</COMPANY>
    <PRICE>10.20</PRICE>
    <YEAR>1990</YEAR>
    <USER_RATING>5</USER_RATING>
  </CD>
  <CD>
    <TITLE>This is US</TITLE>
    <ARTIST>Gary Lee</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>Virgin Records</COMPANY>
    <PRICE>12.20</PRICE>
    <YEAR>1990</YEAR>
    <USER_RATING>2</USER_RATING>
  </CD>
</CATALOG>

```

Notice there is a USER\_RATING element for each CD. Using this data element and the shape manipulation commands, we can create a visual representation of the ratings so that the reader can compare them at a glance.

A template to achieve this is shown in the following figure:

Title	Artist	Rating
<b>F</b> TITLE	<b>A</b> RTIST	<b>E</b> 

The values for the fields are shown in the following table:

Field	Form Field Entry
F	<?for-each:CD?>
TITLE	<?TITLE?>
ARTIST	<?ARTIST?>
E	<?end for-each?>
(star shape)	Web Tab Entry: <pre>&lt;?for-each@shape:xdoxslt:foreach_number(\$_XDOCTX,0,USER_RATING,1)?&gt;</pre> <pre>&lt;?shape-offset-x:(position()-1)*25?&gt;</pre> <pre>&lt;?end for-each?&gt;</pre>

The form fields hold the simple element values. The only difference with this template is the value for the star shape. The replication command is placed in the Web tab of the Format AutoShape dialog.

In the for-each@shape command we are using a command to create a "for...next loop" construct. We specify 1 as the starting number; the value of USER\_RATING as the final number; and 1 as the step value. As the template loops through the CDs, we create an inner loop to repeat a star shape for every USER\_RATING value (that is, a value of 4 will generate 4 stars). The output from this template and the XML sample is shown in the following graphic:

Title	Artist	Rating
Empire Burlesque	Bob Dylan	★★★★
Hide Your Heart	Bonnie Tylor	★★★
Still got the blues	Gary More	★★★★★
This is US	Gary Lee	★★

### Grouped Shape Example

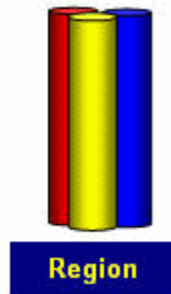
This example shows how to combine shapes into a group and have them react to the incoming data both individually and as a group. Assume the following XML data:

```

<SALES>
  <SALE>
    <REGION>Americas</REGION>
    <SOFTWARE>1200</SOFTWARE>
    <HARDWARE>850</HARDWARE>
    <SERVICES>2000</SERVICES>
  </SALE>
  <SALE>
    <REGION>EMEA</REGION>
    <SOFTWARE>1000</SOFTWARE>
    <HARDWARE>800</HARDWARE>
    <SERVICES>1100</SERVICES>
  </SALE>
  <SALE>
    <REGION>APAC</REGION>
    <SOFTWARE>900</SOFTWARE>
    <HARDWARE>1200</HARDWARE>
    <SERVICES>1500</SERVICES>
  </SALE>
</SALES>

```

You can create a visual representation of this data so that users can very quickly understand the sales data across all regions. Do this by first creating the composite shape in Microsoft Word that you wish to manipulate. The following figure shows a composite shape made up of four components:



The shape consists of three cylinders: red, yellow, and blue. These will represent the data elements software, hardware, and services. The combined object also contains a rectangle that is enabled to receive text from the incoming data.

The following commands are entered into the Web tab:

Red cylinder: <?shape-size-y:SOFTWARE div 1000;'left/bottom'?>

Yellow cylinder: <?shape-size-y:HARDWARE div 1000;'left/bottom'?>

Blue cylinder: <?shape-size-y:SERVICES div 1000;'left/bottom'?>

The shape-size command is used to stretch or shrink the cylinder based on the values of the elements SOFTWARE, HARDWARE, and SERVICES. The value is divided by 1000 to set the stretch or shrink factor. For example, if the value is 2000, divide that by 1000 to get a factor of 2. The shape will generate as twice its current height.

The text-enabled rectangle contains the following command in its Web tab:

<?shape-text:REGION?>

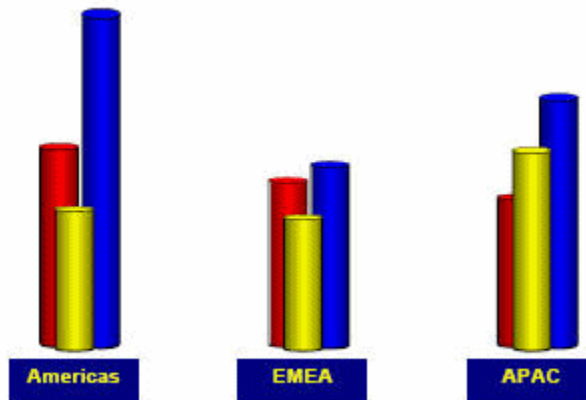
At runtime the value of the REGION element will appear in the rectangle.

All of these shapes were then grouped together and in the Web tab for the grouped object, the following syntax is added:

```
<?for-each@shape:SALE?>  
<?shape-offset-x:(position()-1)*110?>  
<?end for-each?>
```

In this set of commands, the `for-each@shape` loops over the SALE group. The `shape-offset` command moves the next shape in the loop to the right by a specific number of pixels. The expression `(position()-1)` sets the position of the object. The `position()` function returns a record counter while in the loop, so for the first shape, the offset would be  $1-1*100$ , or 0, which would place the first rendering of the object in the position defined in the template. Subsequent occurrences would be rendered at a 100 pixel offset along the x-axis (to the right).

At runtime three sets of shapes will be rendered across the page as shown in the following figure:

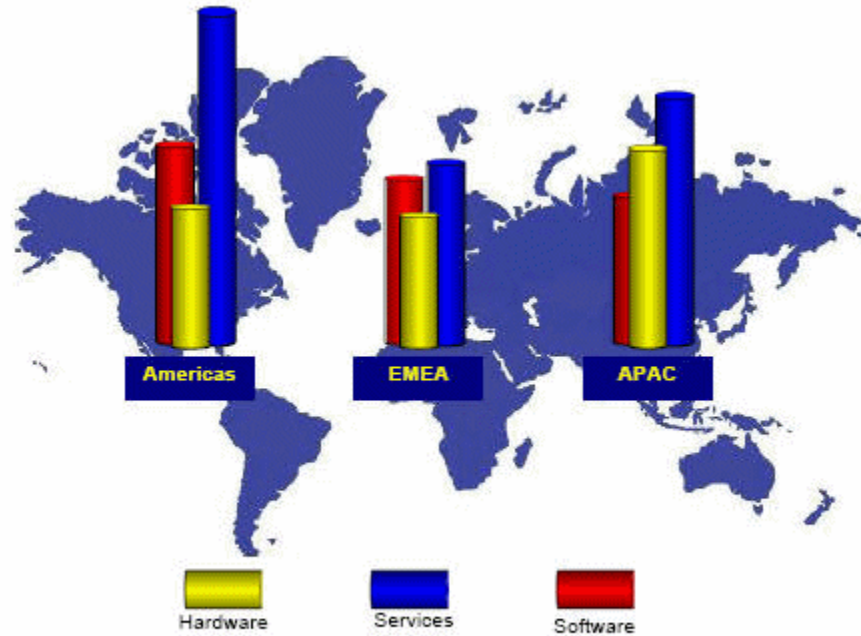


To make an even more visually representative report, these shapes can be superimposed onto a world map. Just use the "Order" dialog in Microsoft Word to layer the map behind the grouped shapes.

**Microsoft Word 2000 Users:** After you add the background map and overlay the shape group, use the Grouping dialog to make the entire composition one group.

**Microsoft Word 2002/3 Users:** These versions of Word have an option under Tools > Options, General tab to "Automatically generate drawing canvas when inserting autosapes". Using this option removes the need to do the final grouping of the map and shapes. We can now generate a visually appealing output for our report as seen in the following figure:





## Supported Native Formatting Features

In addition to the features already listed, XML Publisher supports the following features of Microsoft Word.

### General Features

- Large blocks of text
- Page breaks

To insert a page break, insert a Ctrl-Enter keystroke just before the closing tag of a group. For example if you want the template to start a new page for every Supplier in the Payables Invoice Register:

1. Place the cursor just before the Supplier group's closing `<?end for-each?>` tag.
2. Press Ctrl-Enter to insert a page break.

At runtime each Supplier will start on a new page.

Using this Microsoft Word native feature will cause a single blank page to print at the end of your report output. To avoid this single blank page, use XML Publisher's page break alias. See Special Features: Page Breaks, page 2-49.

- Page numbering

Insert page numbers into your final report by using the page numbering methods of your word processing application. For example, if you are using Microsoft Word:

1. From the **Insert** menu, select **Page Numbers...**
2. Select the **Position**, **Alignment**, and **Format** as desired.

At runtime the page numbers will be displayed as selected.

- Hidden text

You can format text as "hidden" in Microsoft Word and the hidden text will be maintained in RTF output reports.

## Alignment

Use your word processor's alignment features to align text, graphics, objects, and tables.

**Note:** Bidirectional languages are handled automatically using your word processing application's left/right alignment controls.

## Tables

Supported table features include:

- Nested Tables
- Cell Alignment

You can align any object in your template using your word processing application's alignment tools. This alignment will be reflected in the final report output.

- Row spanning and column spanning

You can span both columns and rows in your template as follows:

1. Select the cells you wish to merge.
2. From the **Table** menu, select **Merge Cells**.
3. Align the data within the merged cell as you would normally.

At runtime the cells will appear merged.

- Table Autoformatting

XML Publisher recognizes the table autoformats available in Microsoft Word.

1. Select the table you wish to format.
2. From the **Table** menu, select **Autoformat**.
3. Select the desired table format.

At runtime, the table will be formatted using your selection.

- Cell patterns and colors

You can highlight cells or rows of a table with a pattern or color.

1. Select the cell(s) or table.
2. From the **Table** menu, select **Table Properties**.
3. From the **Table** tab, select the **Borders and Shading...** button.
4. Add borders and shading as desired.

- Repeating table headers

If your data is displayed in a table, and you expect the table to extend across multiple pages, you can define the header rows that you want to repeat at the start of each page.

1. Select the row(s) you wish to repeat on each page.
2. From the **Table** menu, select **Heading Rows Repeat**.

- Prevent rows from breaking across pages.

If you want to ensure that data within a row of a table is kept together on a page, you can set this as an option using Microsoft Word's **Table Properties**.

1. Select the row(s) that you want to ensure do not break across a page.
2. From the **Table** menu, select **Table Properties**.
3. From the **Row** tab, deselect the check box "Allow row to break across pages".

- Fixed-width columns

To set the widths of your table columns:

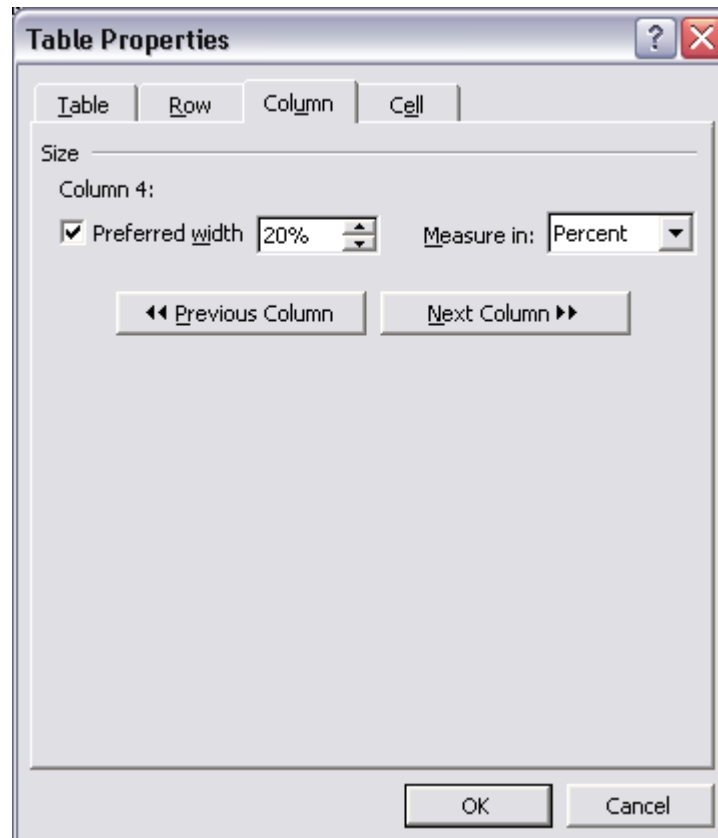
1. Select a column and then select **Table > Table Properties**.
2. In the **Table Properties** dialog, select the **Column** tab.
3. Enable the **Preferred width** checkbox and then enter the width as a **Percent** or

in **Inches**.

4. Select the **Next Column** button to set the width of the next column.

Note that the total width of the columns must add up to the total width of the table.

The following figure shows the **Table Properties** dialog:

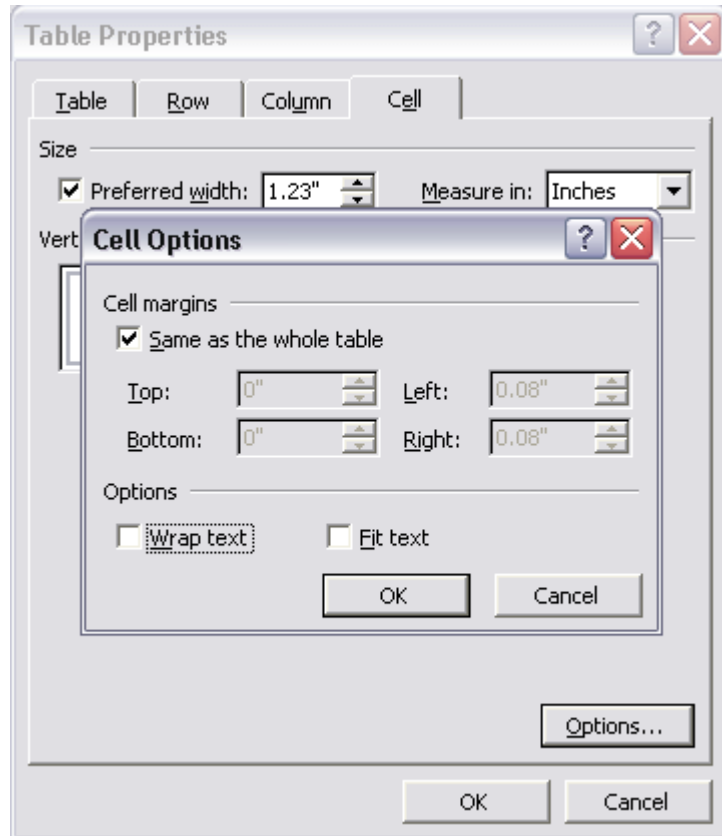


- Text truncation

By default, if the text within a table cell will not fit within the cell, the text will be wrapped. To truncate the text instead, use the table properties dialog.

1. Place your cursor in the cell in which you want the text truncated.
2. Right-click your mouse and select **Table Properties...** from the menu, or navigate to **Table >Table Properties...**
3. From the **Table Properties** dialog, select the **Cell** tab, then select **Options...**
4. Deselect the **Wrap Text** check box.

The following figure shows the Cell Options dialog.



An example of truncation is shown in the following graphic:

Wrap Text checked	Wrap Text unchecked
The quick brown fox jumped over the lazy river.	The quick brown fox

## Date Fields

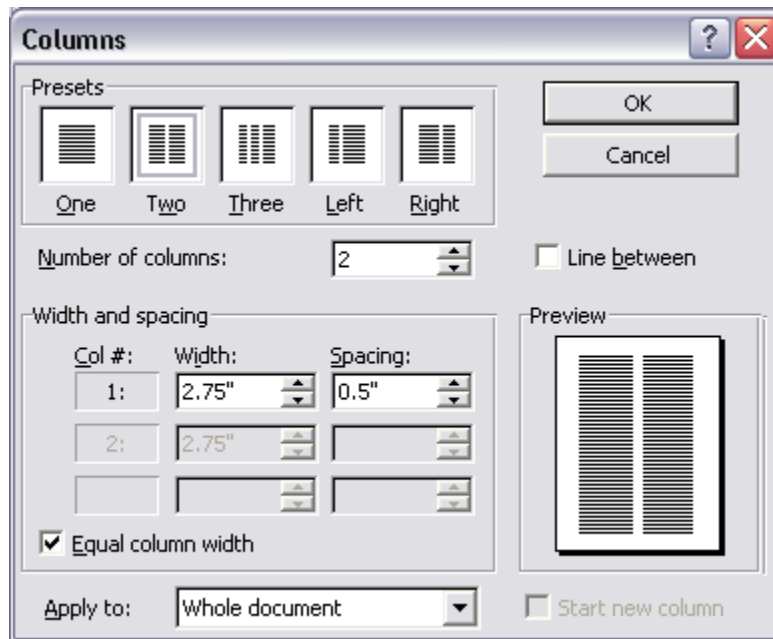
Insert dates using the date feature of your word processing application. Note that this date will correspond to the publishing date, not the request run date.

## Multicolumn Page Support

XML Publisher supports Microsoft Word's Columns function to enable you to publish your output in multiple columns on a page.

Select **Format > Columns** to display the **Columns** dialog box to define the number of

columns for your template. The following graphic shows the Columns dialog:



### Multicolumn Page Example: Labels

To generate address labels in a two-column format:

1. Divide your page into two columns using the Columns command.
2. Define the repeatable group in the first column. Note that you define the repeatable group only in the first column, as shown in the following figure:



Name	CUSTOMER_NAME
Number	CUSTOMER_NUMBER
City	CITY
State	STATE
Zip Code	ZIP_CODE

**Tip:** To prevent the address block from breaking across pages or columns, embed the label block inside a single-celled table. Then specify in the Table Properties that the row should not break across pages. See Prevent rows from breaking across pages, page 2-43.

This template will produce the following multicolumn output:

Name	Nuts and Bolts Ltd	Name	Big Co
Number	1220	Number	1221
City	Espoo	City	Helsinki
State	FI	State	FI
Zip Code	11000000000000000000	Zip Code	11000000000000000000
Name	My Company	Name	Small Co
Number	1220	Number	1221
City	Espoo	City	Helsinki
State	FI	State	FI
Zip Code	11000000000000000000	Zip Code	11000000000000000000

## Background and Watermark Support

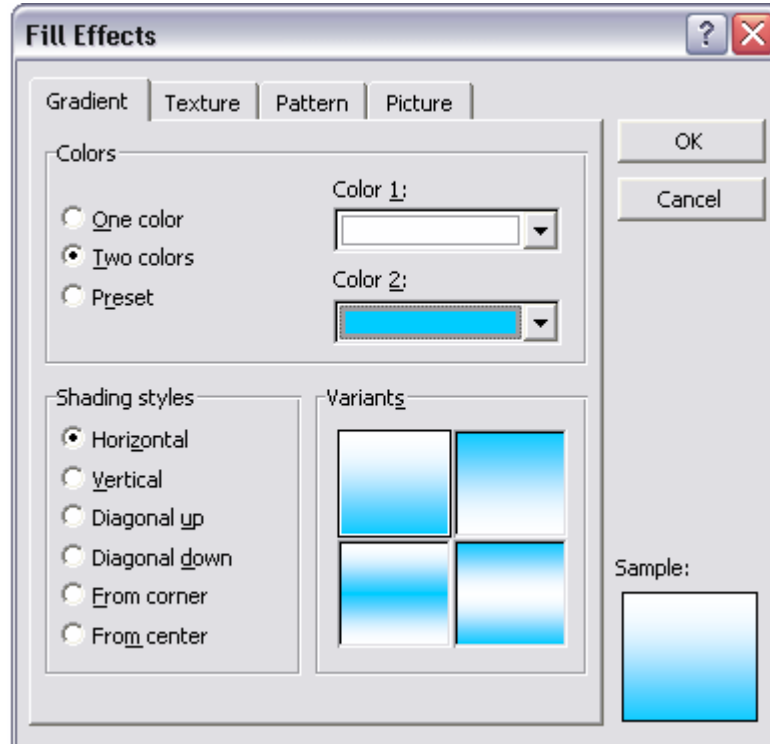
XML Publisher supports the "Background" feature in Microsoft Word. You can specify a single, graduated color or an image background for your template to be displayed in the PDF output. Note that this feature is supported for PDF output only.

To add a background to your template, use the Format > Background menu option.

### Add a Background Using Microsoft Word 2000

From the Background pop up menu, you can:

- Select a single color background from the color palette
- Select Fill Effects to open the Fill Effects dialog. The Fill Effects dialog is shown in the following figure:



From this dialog select one of the following supported options:

- Gradient - this can be either one or two colors
- Texture - choose one of the textures provided, or load your own
- Pattern - select a pattern and background/foreground colors
- Picture - load a picture to use as a background image

#### **Add a Text or Image Watermark Using Microsoft Word 2002 or later**

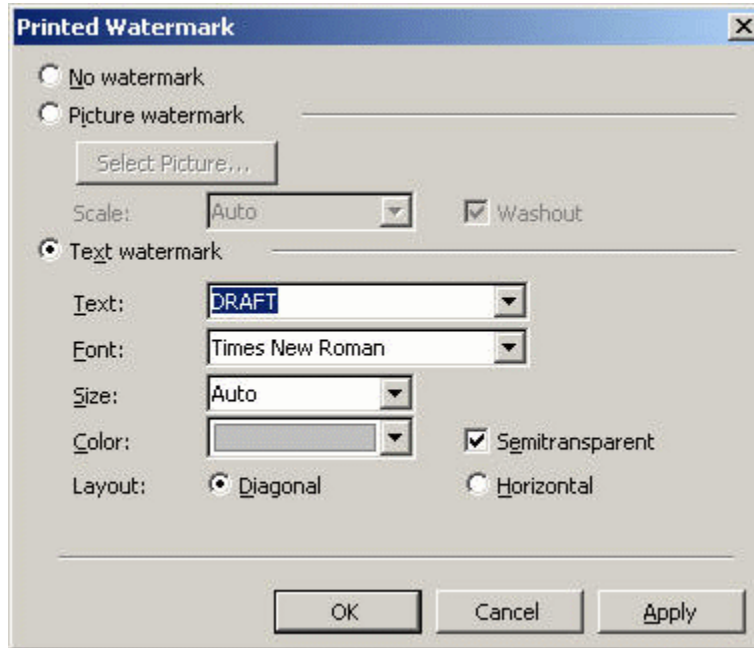
These versions of Microsoft Word allow you to add either a text or image watermark.

Use the Format > Background > Printed Watermark dialog to select either:

- Picture Watermark - load an image and define how it should be scaled on the document
- Text Watermark - use the predefined text options or enter your own, then specify the font, size and how the text should be rendered.

The following figure shows the Printed Watermark dialog completed to display a text watermark:





## Template Features

### Page Breaks

To create a page break after the occurrence of a specific element use the "split-by-page-break" alias. This will cause the report output to insert a hard page break between every instance of a specific element.

To insert a page break between each occurrence of a group, insert the "split-by-page-break" form field within the group immediately before the `<?end for-each?>` tag that closes the group. In the Help Text of this form field enter the syntax:

```
<?split-by-page-break:?>
```

#### Example

For the following XML, assume you want to create a page break for each new supplier:

```

<SUPPLIER>
  <NAME>My Supplier</NAME>
  <INVOICES>
    <INVOICE>
      <INVNUM>10001-1</INVNUM>
      <INVDATE>1-Jan-2005</INVDATE>
      <INVAMT>100</INVOICEAMT>
    </INVOICE>
    <INVOICE>
      <INVNUM>10001-2</INVNUM>
      <INVDATE>10-Jan-2005</INVDATE>
      <INVAMT>200</INVOICEAMT>
    </INVOICE>
  </INVOICES>
</SUPPLIER>
<SUPPLIER>
  <NAME>My Second Supplier</NAME>
  <INVOICES>
    <INVOICE>
      <INVNUM>10001-1</INVNUM>
      <INVDATE>11-Jan-2005</INVDATE>
      <INVAMT>150</INVOICEAMT>
    </INVOICE>
  </INVOICES>
...

```

In the template sample shown in the following figure, the field called PageBreak contains the split-by-page-break syntax:

FE			
Supplier: Supplier 1			
Invoice Number	Invoice Date	Amount	Running Total
FE10001-1	1-Jan-2005	100.00	100.00EFE
PageBreak EFE			

Place the PageBreak field with the `<?split-by-page-break:?>` syntax immediately before the `<?end for-each?>` field. The PageBreak field sits inside the end of the SUPPLIER loop. This will ensure a page break is inserted before the occurrence of each new supplier. This method avoids the ejection of an extra page at the end of the group when using the native Microsoft Word page break after the group.

## Initial Page Number

Some reports require that the initial page number be set at a specified number. For example, monthly reports may be required to continue numbering from month to month. XML Publisher allows you to set the page number in the template to support this requirement.

Use the following syntax in your template to set the initial page number:

```
<?initial-page-number:pagenumber?>
```

where *pagenumber* is the XML element or parameter that holds the numeric value.

### Example 1 - Set page number from XML data element

If your XML data contains an element to carry the initial page number, for example:

```
<REPORT>
  <PAGESTART>200<\PAGESTART>
  . . . .
</REPORT>
```

Enter the following in your template:

```
<?initial-page-number:PAGESTART?>
```

Your initial page number will be the value of the PAGESTART element, which in this case is 200.

### Example 2 - Set page number by passing a parameter value

If you define a parameter called PAGESTART, you can pass the initial value by calling the parameter.

Enter the following in your template:

```
<?initial-page-number:$PAGESTART?>
```

**Note:** You must first declare the parameter in your template. See *Defining Parameters in Your Template*, page 2-89.

## Last Page Only Content

XML Publisher supports the Microsoft Word functionality to specify a different page layout for the first page, odd pages, and even pages. To implement these options, simply select **Page Setup** from the **File** menu, then select the **Layout** tab. XML Publisher will recognize the settings you make in this dialog.

However, Microsoft Word does not provide settings for a different last page only. This is useful for documents such as checks, invoices, or purchase orders on which you may want the content such as the check or the summary in a specific place only on the last page.

XML Publisher provides this ability. To utilize this feature, you must:

1. Create a section break in your template to ensure the content of the final page is separated from the rest of the report.
2. Insert the following syntax on the final page:

```
<?start@last-page:body?>
<?end body?>
```

Any content on the page that occurs above or below these two tags will appear only on the last page of the report. Also, note that because this command explicitly specifies the content of the final page, any desired headers or footers previously defined for the report must be reinserted on the last page.

### Example

This example uses the last page only feature for a report that generates an invoice listing with a summary to appear at the bottom of the last page.

Assume the following XML:

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<INVOICELIST>
  <VENDOR>
    <VENDOR_NAME>Nuts and Bolts Limited</VENDOR_NAME>
    <ADDRESS>1 El Camino Real, Redwood City, CA 94065</ADDRESS>
    <INVOICE>
      <INV_TYPE>Standard</INV_TYPE>
      <INVOICE_NUM>981110</INVOICE_NUM>
      <INVOICE_DATE>10-NOV-04</INVOICE_DATE>
      <INVOICE_CURRENCY_CODE>EUR</INVOICE_CURRENCY_CODE>
      <ENT_AMT>122</ENT_AMT>
      <ACCTD_AMT>122</ACCTD_AMT>
      <VAT_CODE>VAT22%</VAT_CODE>
    </INVOICE>
    <INVOICE>
      <INV_TYPE>Standard</INV_TYPE>
      <INVOICE_NUM>100000</INVOICE_NUM>
      <INVOICE_DATE>28-MAY-04</INVOICE_DATE>
      <INVOICE_CURRENCY_CODE>FIM</INVOICE_CURRENCY_CODE>
      <ENT_AMT>122</ENT_AMT>
      <ACCTD_AMT>20.33</ACCTD_AMT>
      <VAT_CODE>VAT22%</VAT_CODE>
    </INVOICE>
  </VENDOR>
  <VENDOR>
    ...
  <INVOICE>
    ...
  </INVOICE>
</VENDOR>
<SUMMARY>
  <SUM_ENT_AMT>61435</SUM_ENT_AMT>
  <SUM_ACCTD_AMT>58264.68</SUM_ACCTD_AMT>
  <TAX_CODE>EU22%</TAX_CODE>
</SUMMARY>
</INVOICELIST>
```

The report should show each VENDOR and their INVOICE data with a SUMMARY section that appears only on the last page, placed at the bottom of the page. The template for this is shown in the following figure:

### Template Page One

F

Vendor: VENDOR\_NAME

Address: ADDRESS

Invoice Type	Invoice Num	Invoice Date	Invoice Currency	Entered Amount	Accounted Amount
F Invoice	120000	01-Jan-2006	USD	100	100 E

E

<<insert section break>

Insert a Microsoft Word section break (type: next page) on the first page of the template. For the final page, insert new line characters to position the summary table at the bottom of the page. The summary table is shown in the following figure:

### Last Page Only Layout

Last Page Placeholder

#### Tax Summary

Tax Code	Entered Amount	Accounted Amount
F VAT 18.5	100	100E

In this example:

- The F and E components contain the for-each grouping statements.
- The grayed report fields are placeholders for the XML elements.
- The "Last Page Placeholder" field contains the syntax:

```
<?start@last-page:body?><?end body?>
```

to declare the last page layout. Any content above or below this statement will appear on the last page only. The content above the statement is regarded as the header and the content below the statement is regarded as the footer.

If your reports contains headers and footers that you want to carry over onto the last page, you must reinsert them on the last page. For more information about headers and footers see Defining Headers and Footers, page 2-15.

You must insert a section break (type: next page) into the document to specify the last page layout. This example is available in the samples folder of the Oracle BI Publisher Template Builder for Word installation.

It is important to note that if the report is only one page in length, the first page layout will be used. If your report requires that a single page report should default to the last page layout (such as in a check printing implementation) then you can use the following alternate syntax for the "Last Page Placeholder" on the last page:

```
<?start@last-page-first:body?><?end body?>
```

Substituting this syntax will result in the last page layout for reports that are only one page long.

## End on Even or End on Odd Page

If your report has different odd and even page layouts, you may want to force your report to end specifically on an odd or even page. For example, you may include the terms and conditions of a purchase order in the footer of your report using the different odd/even footer functionality (see Different First Page and Different Odd and Even Page Support, page 2-16) and you want to ensure that the terms and conditions are printed on the final page.

Or, you may have binding requirements to have your report end on an even page, without specific layout.

### To end on an even page with layout:

Insert the following syntax in a form field in your template:

```
<?section:force-page-count;'end-on-even-layout'?>
```

### To end on an odd page layout:

```
<?section:force-page-count;'end-on-odd-layout'?>
```

If you do not have layout requirements for the final page, but would like a blank page ejected to force the page count to the preferred odd or even, use the following syntax:

```
<?section:force-page-count;'end-on-even'?>
```

or

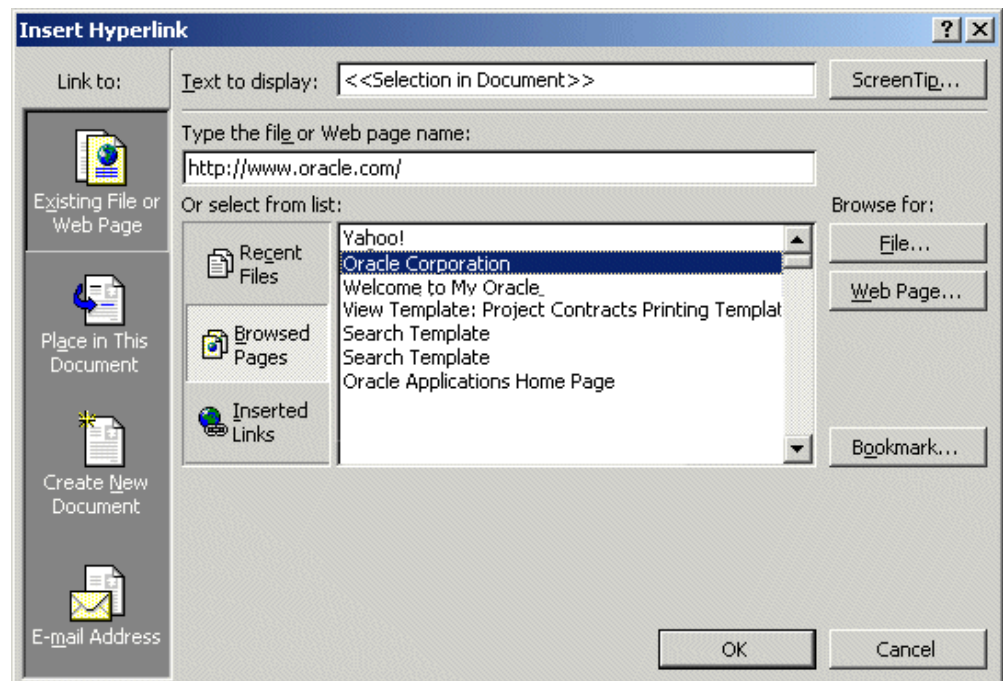
```
<?section:force-page-count;'end-on-odd'?>
```

## Hyperlinks

XML Publisher supports several different types of hyperlinks. The hyperlinks can be fixed or dynamic and can link to either internal or external destinations. Hyperlinks can also be added to shapes.

- To insert static hyperlinks to either text or a shape, use your word processing application's insert hyperlink feature:
  1. Select the text or shape.
  2. Use the right-mouse menu to select **Hyperlink**; or, select **Hyperlink** from the **Insert** menu.
  3. Enter the URL using any of the methods provided on the **Insert Hyperlink** dialog box.

The following screenshot shows the insertion of a static hyperlink using Microsoft Word's **Insert Hyperlink** dialog box.



- If your input XML data includes an element that contains a hyperlink or part of one, you can create dynamic hyperlinks at runtime. In the **Type the file or Web page name** field of the **Insert Hyperlink** dialog box, enter the following syntax:

```
{URL_LINK}
```

where URL\_LINK is the incoming data element name.

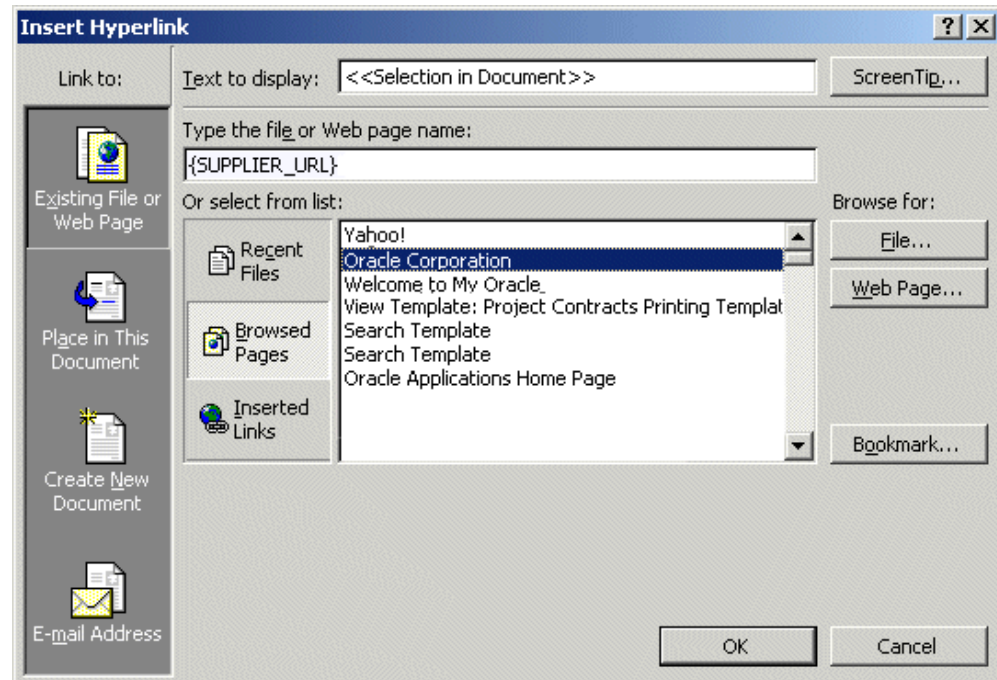
If you have a fixed URL that you want to add elements from your XML data file to construct the URL, enter the following syntax:

```
http://www.oracle.com?product={PRODUCT_NAME}
```

where PRODUCT\_NAME is the incoming data element name.

In both these cases, at runtime the dynamic URL will be constructed.

The following figure shows the insertion of a dynamic hyperlink using Microsoft Word's **Insert Hyperlink** dialog box. The data element SUPPLIER\_URL from the incoming XML file will contain the hyperlink that will be inserted into the report at runtime.



- You can also pass parameters at runtime to construct a dynamic URL.

Enter the parameter and element names surrounded by braces to build up the URL as follows:

```
{$SERVER_URL}{REPORT}/cstid={CUSTOMER_ID}
```

where SERVER\_URL and REPORT are parameters passed to the template at runtime (note the \$ sign) and CUSTOMER\_ID is an XML data element. This link may render as:

```
http://myserver.domain:8888/CustomerReport/cstid=1234
```

### Inserting Internal Links

Insert internal links into your template using Microsoft Word's Bookmark feature.

1. Position your cursor in the desired destination in your document.
2. Select **Insert >Bookmark...**
3. In the **Bookmark** dialog, enter a name for this bookmark, and select **Add**.
4. Select the text or shape in your document that you want to link back to the



Bookmark target.

5. Use the right-mouse menu to select **Hyperlink**; or select **Hyperlink** from the **Insert** menu.
6. On the **Insert Hyperlink** dialog, select **Bookmark**.
7. Choose the bookmark you created from the list.

At runtime, the link will be maintained in your generated report.

## Table of Contents

XML Publisher supports the table of contents generation feature of the RTF specification. Follow your word processing application's procedures for inserting a table of contents.

XML Publisher also provides the ability to create dynamic section headings in your document from the XML data. You can then incorporate these into a table of contents.

To create dynamic headings:

1. Enter a placeholder for the heading in the body of the document, and format it as a "Heading", using your word processing application's style feature. You cannot use form fields for this functionality.

For example, you want your report to display a heading for each company reported. The XML data element tag name is <COMPANY\_NAME>. In your template, enter <?COMPANY\_NAME?> where you want the heading to appear. Now format the text as a Heading.

2. Create a table of contents using your word processing application's table of contents feature.

At runtime the TOC placeholders and heading text will be substituted.

## Generating Bookmarks in PDF Output

If you have defined a table of contents in your RTF template, you can use your table of contents definition to generate links in the Bookmarks tab in the navigation pane of your output PDF. The bookmarks can be either static or dynamically generated.

For information on creating the table of contents, see Table of Contents, page 2-57.

- To create links for a static table of contents:

Enter the syntax:

<?copy-to-bookmark:?>

directly above your table of contents and

```
<?end copy-to-bookmark:?>
```

directly below the table of contents.

- To create links for a dynamic table of contents:

Enter the syntax:

```
<?convert-to-bookmark:?>
```

directly above the table of contents and

```
<?end convert-to-bookmark:?>
```

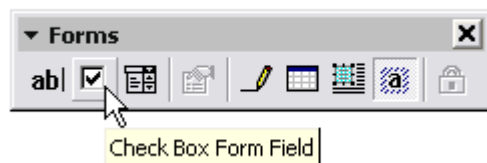
directly below the table of contents.

## Check Boxes

You can include a check box in your template that you can define to display as checked or unchecked based on a value from the incoming data.

To define a check box in your template:

1. Position the cursor in your template where you want the check box to display, and select the Check Box Form Field from the Forms tool bar (shown in the following figure).

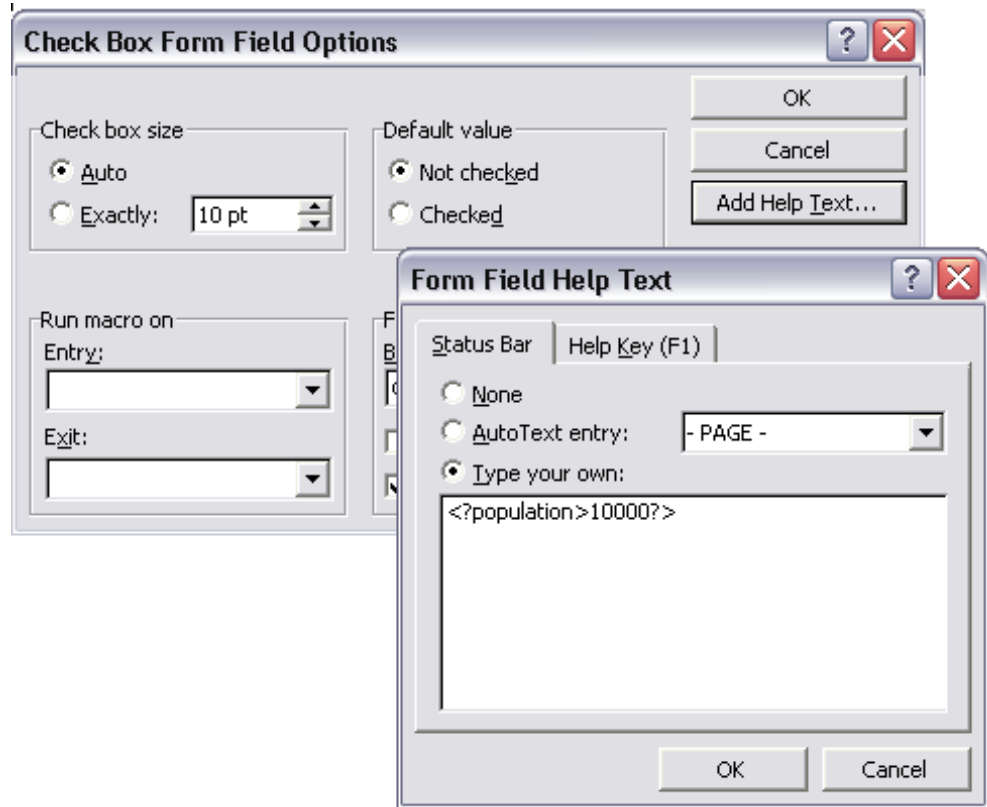


2. Right-click the field to open the **Check Box Form Field Options** dialog.
3. Specify the **Default value** as either Checked or Not Checked.
4. In the Form Field Help Text dialog, enter the criteria for how the box should behave. This must be a boolean expression (that is, one that returns a true or false result).

For example, suppose your XML data contains an element called `<population>`. You want the check box to appear checked if the value of `<population>` is greater than 10,000. Enter the following in the help text field:

```
<?population>10000?>
```

This is displayed in the following figure:



Note that you do not have to construct an "if" statement. The expression is treated as an "if" statement.

See the next section for a sample template using a check box.

## Drop Down Lists

XML Publisher allows you to use the drop-down form field to create a cross-reference in your template from your XML data to some other value that you define in the drop-down form field.

For example, suppose you have the following XML:

```

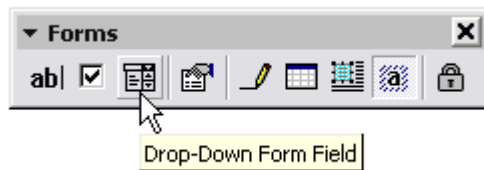
<countries>
  <country>
    <name>Chad</name>
    <population>7360000</population>
    <continentIndex>5</continentIndex>
  </country>
  <country>
    <name>China</name>
    <population>1265530000</population>
    <continentIndex>1</continentIndex>
  </country>
  <country>
    <name>Chile</name>
    <population>14677000</population>
    <continentIndex>3</continentIndex>
  </country>
  . . .
</countries>

```

Notice that each `<country>` entry has a `<continentindex>` entry, which is a numeric value to represent the continent. Using the drop-down form field, you can create an index in your template that will cross-reference the `<continentindex>` value to the actual continent name. You can then display the name in your published report.

To create the index for the continent example:

1. Position the cursor in your template where you want the value from the drop-down list to display, and select the Drop-Down Form Field from the Forms tool bar (shown in the following figure).



2. Right-click the field to display the **Drop-Down Form Field Options** dialog.
3. Add each value to the **Drop-down item** field and the click **Add** to add it to the **Items in drop-down list** group. The values will be indexed starting from one for the first, and so on. For example, the list of continents will be stored as follows:

Index	Value
1	Asia
2	North America

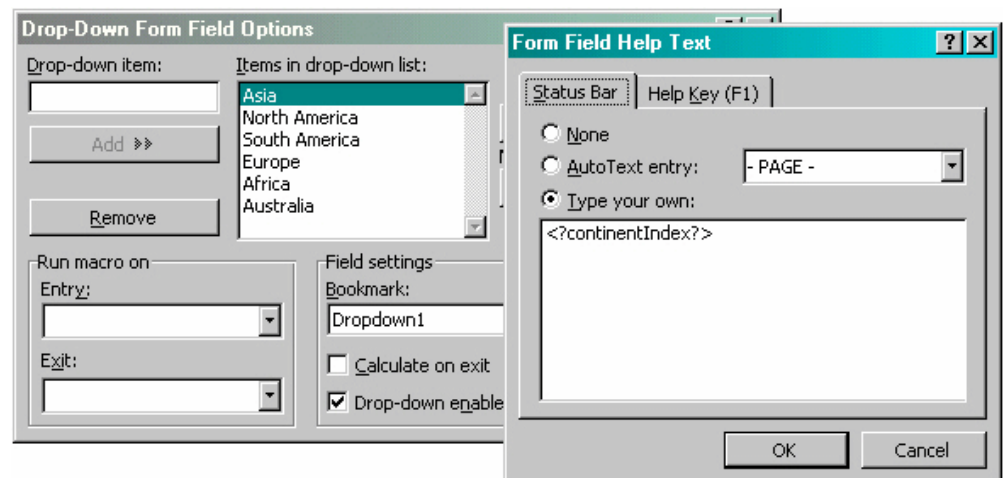
Index	Value
3	South America
4	Europe
5	Africa
6	Australia

- Now use the Help Text box to enter the XML element name that will hold the index for the drop-down field values.

For this example, enter

```
<?continentIndex?>
```

The following figure shows the **Drop-Down Form Field Options** dialogs for this example:



Using the check box and drop-down list features, you can create a report to display population data with check boxes to demonstrate figures that reach a certain limit. An example is shown in the following figure:

Country	Population	more than 10M?	Continent
Chad	7,360,000	<input type="checkbox"/>	Africa
China	1,265,530,000	<input checked="" type="checkbox"/>	Asia
Chile	14,677,000	<input checked="" type="checkbox"/>	South America
Sweden	8,887,000	<input type="checkbox"/>	Europe
United States	270,312,000	<input checked="" type="checkbox"/>	North America
New Zealand	3,625,000	<input type="checkbox"/>	Australia

The template to create this report is shown in the next figure:

Country	Population	more than 10M?	Continent
FE China	1,000,000	<input type="checkbox"/>	Asia EFE

where the fields have the following values:

Field	Form Field Entry	Description
FE	<?for-each:country?>	Begins the <code>country</code> repeating group.
China	<?name?>	Placeholder for the <code>name</code> element.
1,000,000	<?population?>	Placeholder for the <code>population</code> element.
(check box)	<?population>1000000?>	Establishes the condition for the check box. If the value for the <code>population</code> element is greater than 1,000,000, the check box will display as checked.
Asia	<?continentIndex?>	The drop-down form field for the <code>continentIndex</code> element. See the preceding description for its contents. At runtime, the value of the XML element is replaced with the value it is cross-referenced to in the drop-down form field.
EFE	<?end for-each?>	Ends the <code>country</code> group.

## Conditional Formatting

Conditional formatting occurs when a formatting element appears only when a certain condition is met. XML Publisher supports the usage of simple "if" statements, as well as more complex "choose" expressions.

The conditional formatting that you specify can be XSL or XSL:FO code, or you can specify actual RTF objects such as a table or data. For example, you can specify that if reported numbers reach a certain threshold, they will display shaded in red. Or, you

can use this feature to hide table columns or rows depending on the incoming XML data.

## If Statements

Use an if statement to define a simple condition; for example, if a data field is a specific value.

1. Insert the following syntax to designate the beginning of the conditional area.

```
<?if:condition?>
```

2. Insert the following syntax at the end of the conditional area: `<?end if?>`.

For example, to set up the Payables Invoice Register to display invoices only when the Supplier name is "Company A", insert the syntax `<?if:VENDOR_NAME='COMPANY A'?>` before the Supplier field on the template.

Enter the `<?end if?>` tag after the invoices table.

This example is displayed in the figure below. Note that you can insert the syntax in form fields, or directly into the template.

Group: Suppliers

`<?if:VENDOR_NAME='Company A'?>`

Supplier: **Supplier 1**

Invoice Num	Invoice Date
Group:Invoices 1234566	1-Jan-2004

`<?end if?>`

End:Suppliers

## If Statements in Boilerplate Text

Assume you want to incorporate an "if" statement into the following free-form text:

The program was (not) successful.

You only want the "not" to display if the value of an XML tag called `<SUCCESS>` equals "N".

To achieve this requirement, you must use the XML Publisher context command to

place the if statement into the inline sequence rather than into the block (the default placement).

**Note:** For more information on context commands, see Using Context Commands, page 2-123.

For example, if you construct the code as follows:

```
The program was <?if:SUCCESS='N'?>not<?end if?> successful.
```

The following undesirable result will occur:

```
The program was
not
successful.
```

because XML Publisher applies the instructions to the block by default. To specify that the if statement should be inserted into the inline sequence, enter the following:

```
The program was <?if@inlines:SUCCESS='N'?>not<?end if?>
successful.
```

This construction will result in the following display:

```
The program was successful.
```

If SUCCESS does not equal 'N';

or

```
The program was not successful.
```

If SUCCESS equals 'N'.

## If-then-Else Statements

XML Publisher supports the common programming construct "if-then-else". This is extremely useful when you need to test a condition and conditionally show a result. For example:

```
IF X=0 THEN
  Y=2
ELSE
  Y=3
END IF
```

You can also nest these statements as follows:

```
IF X=0 THEN
  Y=2
ELSE
  IF X=1 THEN
    Y=10
  ELSE Y=100
END IF
```

Use the following syntax to construct an if-then-else statement in your RTF template:

```
<?xdofx:if element_condition then result1 else result2 end if?>
```



For example, the following statement tests the AMOUNT element value. If the value is greater than 1000, show the word "Higher"; if it is less than 1000, show the word "Lower"; if it is equal to 1000, show "Equal":

```
<?xdofx:if AMOUNT > 1000 then 'Higher'
  else
    if AMOUNT < 1000 then 'Lower'
    else
      'Equal'
    end if?>
```

## Choose Statements

Use the `choose`, `when`, and `otherwise` elements to express multiple conditional tests. If certain conditions are met in the incoming XML data then specific sections of the template will be rendered. This is a very powerful feature of the RTF template. In regular XSL programming, if a condition is met in the `choose` command then further XSL code is executed. In the template, however, you can actually use visual widgets in the conditional flow (in the following example, a table).

Use the following syntax for these elements:

```
<?choose:?>
<?when:expression?>
<?otherwise?>
```

### "Choose" Conditional Formatting Example

This example shows a `choose` expression in which the display of a row of data depends on the value of the fields `EXEMPT_FLAG` and `POSTED_FLAG`. When the `EXEMPT_FLAG` equals "^", the row of data will render light gray. When `POSTED_FLAG` equals "\*" the row of data will render shaded dark gray. Otherwise, the row of data will render with no shading.

In the following figure, the form field default text is displayed. The form field help text entries are shown in the table following the example.

<b>Column Legend:</b>	<div> <div></div> 'Not Posted' <div></div> 'Reduces Available Exemption Limit' </div>
-----------------------	---

	Tax Code	Taxable Recoverable	Taxable Non-Recoverable	Recoverable Tax	Tax Non-Recoverable	Total
e	<Grp:VAT					
	<Choose					
	<When EXEMPT_FLAG='^'					
	VAT 15%	1000	1000	1000	1000	1000
	End When>					
	<When POSTED_FLAG='^'					
	VAT 15%	1000	1000	1000	1000	1000
	End When>					
	Otherwise					
	VAT 15%	1000	1000	1000	1000	1000]
End Otherwise>]						
End Choose>						
End VAT>						

Default Text Entry in Example Form Field	Help Text Entry in Form Field
<Grp:VAT	<?for-each:VAT?>
<Choose	<?choose?>
<When EXEMPT_FLAG='^'	<?When EXEMPT_FLAG='^'?>
End When>	<?end When?>
<When EXEMPT_FLAG='^'	<?When EXEMPT_FLAG='^'?>
End When>	<?end When?>

Column Formatting

You can conditionally show and hide columns of data in your document output. The following example demonstrates how to set up a table so that a column is only displayed based on the value of an element attribute.

This example will show a report of a price list, represented by the following XML:

```

<items type="PUBLIC"> <!-- can be marked 'PRIVATE' -->
  <item>
    <name>Plasma TV</name>
    <quantity>10</quantity>
    <price>4000</price>
  </item>
  <item>
    <name>DVD Player</name>
    <quantity>3</quantity>
    <price>300</price>
  </item>
  <item>
    <name>VCR</name>
    <quantity>20</quantity>
    <price>200</price>
  </item>
  <item>
    <name>Receiver</name>
    <quantity>22</quantity>
    <price>350</price>
  </item>
</items>

```

Notice the type attribute associated with the items element. In this XML it is marked as "PUBLIC" meaning the list is a public list rather than a "PRIVATE" list. For the "public" version of the list we do not want to show the quantity column in the output, but we want to develop only one template for both versions based on the list type.

The following figure is a simple template that will conditionally show or hide the quantity column:

Name	<b>IF</b> Quantity <b>end-if</b>	Price
grp:ItemPlasma TV	20	1,000.00end grp

The following table shows the entries made in the template for the example:

Default Text	Form Field Entry	Description
grp:Item	<?for-each:item?>	Holds the opening for-each loop for the item element.
Plasma TV	<?name?>	The placeholder for the name element from the XML file.

Default Text	Form Field Entry	Description
IF	<?if@column:/items/@type="PRIVATE"?>	The opening of the if statement to test for the attribute value "PRIVATE". Note that this syntax uses an XPath expression to navigate back to the "items" level of the XML to test the attribute. For more information about using XPath in your templates, see XPath Overview, page 2-119.
Quantity	N/A	Boilerplate heading
end-if	<?end if?>	Ends the if statement.
20	<?if@column:/items/@type="PRIVATE"?><?quantity?><?end if?>	The placeholder for the quantity element surrounded by the "if" statement.
1,000.00	<?price?>	The placeholder for the price element.
end grp	<?end for-each?>	Closing tag of the for-each loop.

The conditional column syntax is the "if" statement syntax with the addition of the @column clause. It is the @column clause that instructs XML Publisher to hide or show the column based on the outcome of the if statement.

If you did not include the @column the data would not display in your report as a result of the if statement, but the column still would because you had drawn it in your template.

**Note:** The @column clause is an example of a context command. For more information, see Using Context Commands, page 2-123.

The example will render the output shown in the following figure:

Name	Price
Plasma TV	4,000.00
DVD Player	300.00
VCR	200.00
Receiver	350.00

If the same XML data contained the type attribute set to "PRIVATE" the following output would be rendered from the same template:

Name	Quantity	Price
Plasma TV	10	4,000.00
DVD Player	3	300.00
VCR	20	200.00
Receiver	22	350.00

## Row Formatting

XML Publisher allows you to specify formatting conditions as the row-level of a table. Examples of row-level formatting are:

- Highlighting a row when the data meets a certain threshold.
- Alternating background colors of rows to ease readability of reports.
- Showing only rows that meet a specific condition.

### Conditionally Displaying a Row

To display only rows that meet a certain condition, insert the `<?if:condition?>` `<?end if?>` tags at the beginning and end of the row, within the for-each tags for the group. This is demonstrated in the following sample template.

Industry	Year	Month	Sales
<code>for-each SALE if big INDUSTRY</code>	<code>YEAR</code>	<code>MONTH</code>	<code>SALES end if end SALE</code>

Note the following fields from the sample figure:

Default Text Entry	Form Field Help Text	Description
for-each SALE	<code>&lt;?for-each:SALE?&gt;</code>	Opens the for-each loop to repeat the data belonging to the SALE group.
if big	<code>&lt;?if:SALES&gt;5000?&gt;</code>	If statement to display the row only if the element SALES has a value greater than 5000.
INDUSTRY	<code>&lt;?INDUSTRY?&gt;</code>	Data field
YEAR	<code>&lt;?YEAR?&gt;</code>	Data field

Default Text Entry	Form Field Help Text	Description
MONTH	<?MONTH?>	Data field
SALES end if	<?end if?>	Closes the if statement.
end SALE	<?end for-each?>	Closes the SALE loop.

### Conditionally Highlighting a Row

This example demonstrates how to set a background color on every other row. The template to create this effect is shown in the following figure:

Industry	Year	Month	Sales
for-each SALE format; INDUSTRY	YEAR	MONTH	SALES end SALE

The following table shows values of the form fields in the template:

Default Text Entry	Form Field Help Text	Description
for-each SALE	<?for-each:SALE?>	Defines the opening of the for-each loop for the SALE group.
format;	<?if@row:position() mod 2=0?> <xsl:attribute name="background-color" xdofo:ctx="incontext">lightgray</xsl:attribute><?end if?>	For each alternate row, the background color attribute is set to gray for the row.
INDUSTRY	<?INDUSTRY?>	Data field
YEAR	<?YEAR?>	Data field
MONTH	<?MONTH?>	Data field
SALES	<?SALES?>	Data field
end SALE	<?end for-each?>	Closes the SALE for-each loop.

In the preceding example, note the "format;" field. It contains an if statement with a "row" context (@row). This sets the context of the if statement to apply to the current row. If the condition is true, then the <xsl:attribute> for the background color of the row will be set to light gray. This will result in the following output:

Industry	Year	Month	Sales
Oil	2000	Jan	100,000
Automotive	2000	Jan	200,000
Groceries	2000	Jan	50,000

**Note:** For more information about context commands, see Using Context Commands, page 2-123.

## Cell Highlighting

The following example demonstrates how to conditionally highlight a cell based on a value in the XML file.

For this example we will use the following XML:

```
<accounts>
  <account>
    <number>1-100-3333</number>
    <debit>100</debit>
    <credit>300</credit>
  </account>
  <account>
    <number>1-101-3533</number>
    <debit>220</debit>
    <credit>30</credit>
  </account>
  <account>
    <number>1-130-3343</number>
    <debit>240</debit>
    <credit>1100</credit>
  </account>
  <account>
    <number>1-153-3033</number>
    <debit>3000</debit>
    <credit>300</credit>
  </account>
</accounts>
```

The template lists the accounts and their credit and debit values. In the final report we want to highlight in red any cell whose value is greater than 1000. The template for this is shown in the following figure:

Account	Debit	Credit
FE:Account1-232-4444	CH1100.00	CH2100.00EFE

The field definitions for the template are shown in the following table:

Default Text Entry	Form Field Entry	Description
FE:Account	<?for-each:account?>	Opens the for each-loop for the element <code>account</code> .
1-232-4444	<?number?>	The placeholder for the <code>number</code> element from the XML file.
CH1	<?if:debit>1000?><xsl:attribute xdofo:ctx="block" name="background-color">red</xsl:attribute><?end if?>	This field holds the code to highlight the cell red if the debit amount is greater than 1000.
100.00	<?debit?>	The placeholder for the <code>debit</code> element.
CH2	<?if:credit>1000?><xsl:attribute xdofo:ctx="block" name="background-color">red</xsl:attribute><?end if?>	This field holds the code to highlight the cell red if the credit amount is greater than 1000.
100.00	<?credit?>	The placeholder for the <code>credit</code> element.
EFE	<?end for-each?>	Closes the for-each loop.

The code to highlight the debit column as shown in the table is:

```
<?if:debit>1000?>
  <xsl:attribute
    xdofo:ctx="block" name="background-color">red
  </xsl:attribute>
<?end if?>
```

The "if" statement is testing if the debit value is greater than 1000. If it is, then the next lines are invoked. Notice that the example embeds native XSL code inside the "if" statement.

The "attribute" element allows you to modify properties in the XSL.

The `xdofo:ctx` component is an XML Publisher feature that allows you to adjust XSL attributes at any level in the template. In this case, the background color attribute is changed to red.

To change the color attribute, you can use either the standard HTML names (for example, red, white, green) or you can use the hexadecimal color definition (for example, #FFFFFF).

The output from this template is displayed in the following figure:



Account	Debit	Credit
1-100-3333	100.00	300.00
1-101-3533	220.00	30.00
1-130-3343	240.00	1100.00
1-153-3033	3000.00	300.00

## Page-Level Calculations

### Displaying Page Totals

XML Publisher allows you to display calculated page totals in your report. Because the page is not created until publishing time, the totaling function must be executed by the formatting engine.

**Note:** Page totaling is performed in the PDF-formatting layer. Therefore this feature is not available for other outputs types: HTML, RTF, Excel.

**Note:** Note that this page totaling function will only work if your source XML has raw numeric values. The numbers must not be preformatted.

Because the page total field does not exist in the XML input data, you must define a variable to hold the value. When you define the variable, you associate it with the element from the XML file that is to be totaled for the page. Once you define total fields, you can also perform additional functions on the data in those fields.

To declare the variable that is to hold your page total, insert the following syntax immediately following the placeholder for the element that is to be totaled:

```
<?add-page-total:TotalFieldName;'element'??>
```

where

`TotalFieldName` is the name you assign to your total (to reference later) and

`'element'` is the XML element field to be totaled.

You can add this syntax to as many fields as you want to total.

Then when you want to display the total field, enter the following syntax:

```
<?show-page-total:TotalFieldName;'Oracle-number-format'??>
```

where

`TotalFieldName` is the name you assigned to give the page total field above and

`Oracle-number-format` is the format you wish to use to for the display, using the Oracle format mask (for example: C9G999D00). For the list of Oracle format mask

symbols, see Using the Oracle Format Mask, page 2-111.

The following example shows how to set up page total fields in a template to display total credits and debits that have displayed on the page, and then calculate the net of the two fields.

This example uses the following XML:

```
<balance_sheet>
  <transaction>
    <debit>100</debit>
    <credit>90</credit>
  </transaction>
  <transaction>
    <debit>110</debit>
    <credit>80</credit>
  </transaction>
  ...
</balance_sheet>
```

The following figure shows the table to insert in the template to hold the values:

Debit	Credit
FE 100.00	90.00 Net EFE

The following table shows the form field entries made in the template for the example table:

Default Text Entry	Form Field Help Text Entry	Description
FE	<?for-each:transaction?>	This field defines the opening "for-each" loop for the transaction group.
100.00	<?debit?><?add-page-total:dt;'debit'?'>	This field is the placeholder for the debit element from the XML file. Because we want to total this field by page, the page total declaration syntax is added. The field defined to hold the total for the debit element is dt.
90.00	<?credit?><?add-page-total:ct;'credit'?'>	This field is the placeholder for the credit element from the XML file. Because we want to total this field by page, the page total declaration syntax is added. The field defined to hold the total for the credit element is ct.

Default Text Entry	Form Field Help Text Entry	Description
Net	<code>&lt;add-page-total:net;'debit - credit'??&gt;</code>	Creates a net page total by subtracting the credit values from the debit values.
EFE	<code>&lt;?end for-each?&gt;</code>	Closes the for-each loop.

Note that on the field defined as "net" we are actually carrying out a calculation on the values of the `credit` and `debit` elements.

Now that you have declared the page total fields, you can insert a field in your template where you want the page totals to appear. Reference the calculated fields using the names you supplied (in the example, `ct` and `dt`). The syntax to display the page totals is as follows:

For example, to display the debit page total, enter the following:

```
<?show-page-total:dt;'C9G990D00';'(C9G990D00)'??>
```

Therefore to complete the example, place the following at the bottom of the template page, or in the footer:

```
Page Total Debit: <?show-page-total:dt;'C9G990D00';'(C9G990D00)'??>
```

```
Page Total Credit: <?show-page-total:ct;'C9G990D00';'(C9G990D00)'??>
```

```
Page Total Balance: <?show-page-total:net;'C9G990D00';'(C9G990D00)'??>
```

The output for this report is shown in the following graphic:

		Page 1
Debit	Credit	
100.00	90.00	
110.00	80.00	
120.00	70.00	
130.00	60.00	
140.00	50.00	
150.00	40.00	
		Page Total Debit:750.00
		Page Total Credit:390.00
		Page Total Balance:360.00

## Brought Forward/Carried Forward Totals

Many reports require that a page total be maintained throughout the report output and be displayed at the beginning and end of each page. These totals are known as "brought

forward/carried forward" totals.

**Note:** The totaling for the brought forward and carried forward fields is performed in the PDF-formatting layer. Therefore this feature is not available for other outputs types: HTML, RTF, Excel.

An example is displayed in the following figure:

Page 1			Page 2 Brought Forward: 300			Page 3 Brought Forward: 600		
Inv	Date	Amount	Inv	Date	Amount	Inv	Date	Amount
1001	1-Jan-05	100	1004	1-Jan-05	100	1007	1-Jan-05	100
1002	1-Jan-05	100	1005	1-Jan-05	100	1008	1-Jan-05	100
1003	1-Jan-05	100	1006	1-Jan-05	100	1009	1-Jan-05	100
Carried Forward: 300			Carried Forward: 600					

At the end of the first page, the page total for the Amount element is displayed as the Carried Forward total. At the top of the second page, this value is displayed as the Brought Forward total from the previous page. At the bottom of the second page, the brought forward value plus the total for that page is calculated and displayed as the new Carried Forward value, and this continues throughout the report.

This functionality is an extension of the Page Totals, page 2-73 feature. The following example walks through the syntax and setup required to display the brought forward and carried forward totals in your published report.

Assume you have the following XML:

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<INVOICES>
  <INVOICE>
    <INVNUM>10001-1</INVNUM>
    <INVDATE>1-Jan-2005</INVDATE>
    <INVAMT>100</INVOICEAMT>
  </INVOICE>
  <INVOICE>
    <INVNUM>10001-2</INVNUM>
    <INVDATE>10-Jan-2005</INVDATE>
    <INVAMT>200</INVOICEAMT>
  </INVOICE>
  <INVOICE>
    <INVNUM>10001-1</INVNUM>
    <INVDATE>11-Jan-2005</INVDATE>
    <INVAMT>150</INVOICEAMT>
  </INVOICE>
  .
  .
  .
</INVOICES>
```

The following sample template creates the invoice table and declares a placeholder that will hold your page total:

Init PTs

Invoice	Date	Amount
FE 132342	10-May-2005	1,000.00 InvAmt EG

End PTs

The fields in the template have the following values:

Field	Form Field Help Text Entry	Description
Init PTs	<?init-page-total: InvAmt?>	Declares "InvAmt" as the placeholder that will hold the page total.
FE	<?for-each:INVOICE?>	Begins the INVOICE group.
10001-1	<?INVNUM?>	Placeholder for the Invoice Number tag.
1-Jan-2005	<?INVDATE?>	Placeholder for the Invoice Date tag.
100.00	<?INVAMT?>	Placeholder for the Invoice Amount tag.
InvAmt	<?add-page-total:InvAmt;INVAMT?>	Assigns the "InvAmt" page total object to the INVAMT element in the data.
EFE	<?end for-each?>	Closes the INVOICE group.
End PTs	<?end-page-total:InvAmt?>	Closes the "InvAmt" page total.

To display the brought forward total at the top of each page (except the first), use the following syntax:

```
<xdofo:inline-total
  display-condition="exceptfirst"
  name="InvAmt">
  Brought Forward:
<xdofo:show-brought-forward
  name="InvAmt"
  format="99G999G999D00"/>
</xdofo:inline-total>
```

The following table describes the elements comprising the brought forward syntax:

Code Element	Description and Usage
<code>inline-total</code>	<p>This element has two properties:</p> <ul style="list-style-type: none"> <li><code>name</code> - name of the variable you declared for the field.</li> <li><code>display-condition</code> - sets the display condition. This is an optional property that takes one of the following values: <ul style="list-style-type: none"> <li><code>first</code> - the contents appear only on the first page</li> <li><code>last</code> - the contents appear only on the last page</li> <li><code>exceptfirst</code> - contents appear on all pages except first</li> <li><code>exceptlast</code> - contents appear on all pages except last</li> <li><code>everytime</code> - (default) contents appear on every page</li> </ul> </li> </ul> <p>In this example, <code>display-condition</code> is set to "exceptfirst" to prevent the value from appearing on the first page where the value would be zero.</p>
<code>Brought Forward:</code>	<p>This string is optional and will display as the field name on the report.</p>
<code>show-brought-forward</code>	<p>Shows the value on the page. It has the following two properties:</p> <ul style="list-style-type: none"> <li><code>name</code> - the name of the field to show. In this case, "InvAmt". This property is mandatory.</li> <li><code>format</code> - the Oracle number format to apply to the value at runtime. This property is optional, but if you want to supply a format mask, you must use the Oracle format mask. For more information, see Using the Oracle Format Mask, page 2-111 .</li> </ul>

Insert the brought forward object at the top of the template where you want the brought forward total to display. If you place it in the body of the template, you can insert the syntax in a form field.

If you want the brought forward total to display in the header, you must insert the full code string into the header because Microsoft Word does not support form fields in the header or footer regions. However, you can alternatively use the start body/end body syntax which allows you to define what the body area of the report will be. XML Publisher will recognize any content above the defined body area as header content, and any content below as the footer. This allows you to use form fields. See Multiple or

Complex Headers and Footers, page 2-16 for details.

Place the carried forward object at the bottom of your template where you want the total to display. The carried forward object for our example is as follows:

```
<xdofo:inline-total
  display-condition="exceptlast"
  name="InvAmt">
  Carried Forward:
<xdofo:show-carry-forward
  name="InvAmt"
  format="99G999G999D00"/>
</xdofo:inline-total>
```

Note the following differences with the brought-forward object:

- The `display-condition` is set to `exceptlast` so that the carried forward total will display on every page except the last page.
- The display string is "Carried Forward".
- The `show-carry-forward` element is used to show the carried forward value. It has the same properties as `brought-carried-forward`, described above.

You are not limited to a single value in your template, you can create multiple brought forward/carried forward objects in your template pointing to various numeric elements in your data.

## Running Totals

### Example

The variable functionality (see Using Variables, page 2-88) can be used to add a running total to your invoice listing report. This example assumes the following XML structure:

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<INVOICES>
  <INVOICE>
    <INVNUM>10001-1</INVNUM>
    <INVDATE>1-Jan-2005</INVDATE>
    <INVAMT>100</INVOICEAMT>
  </INVOICE>
  <INVOICE>
    <INVNUM>10001-2</INVNUM>
    <INVDATE>10-Jan-2005</INVDATE>
    <INVAMT>200</INVOICEAMT>
  </INVOICE>
  <INVOICE>
    <INVNUM>10001-1</INVNUM>
    <INVDATE>11-Jan-2005</INVDATE>
    <INVAMT>150</INVOICEAMT>
  </INVOICE>
</INVOICES>
```

Using this XML, we want to create the report that contains running totals as shown in the following figure:

Invoice Number	Invoice Date	Amount	Running Total
1000-1	1-Jan-2005	100.00	100.00
1000-2	10-Jan-2005	200.00	300.00
1000-3	11-Jan-2005	150.00	450.00

To create the Running Total field, define a variable to track the total and initialize it to 0. The template is shown in the following figure:

RTotalVar			
Invoice Number	Invoice Date	Amount	Running Total
FE10001-1	1-Jan-2005	100.00	100.00EFE

The values for the form fields in the template are shown in the following table:

Form Field	Syntax	Description
RtotalVar	<?xdoxslt:set_variable(\$_XDO CTX, 'RTotalVar', 0)?>	Declares the "RTotalVar" variable and initializes it to 0.
FE	<?for-each:INVOICE?>	Starts the Invoice group.
10001-1	<?INVNUM?>	Invoice Number tag
1-Jan-2005	<?INVDATE?>	Invoice Date tag
100.00	<?xdoxslt:set_variable(\$_XDO CTX, 'RTotalVar', xdoxslt:get_variable(\$_XDOC TX, 'RTotalVar') + INVAMT)?>  xdoxslt:get_variable(\$_XDOC TX, 'RTotalVar')?>	Sets the value of RTotalVar to the current value plus the new Invoice Amount.  Retrieves the RTotalVar value for display.
EFE	<?end for-each?>	Ends the INVOICE group.



## Data Handling

### Sorting

You can sort a group by any element within the group. Insert the following syntax within the group tags:

```
<?sort:element name?>
```

For example, to sort the Payables Invoice Register (shown at the beginning of this chapter) by Supplier (VENDOR\_NAME), enter the following after the

```
<?for-each:G_VENDOR_NAME?> tag:
```

```
<?sort:VENDOR_NAME?>
```

To sort a group by multiple fields, just insert the sort syntax after the primary sort field. To sort by Supplier and then by Invoice Number, enter the following

```
<?sort:VENDOR_NAME?> <?sort:INVOICE_NUM?>
```

### Checking for Nulls

Within your XML data there are three possible scenarios for the value of an element:

- The element is present in the XML data, and it has a value
- The element is present in the XML data, but it does not have a value
- The element is not present in the XML data, and therefore there is no value

In your report layout, you may want to specify a different behavior depending on the presence of the element and its value. The following examples show how to check for each of these conditions using an "if" statement. The syntax can also be used in other conditional formatting constructs.

- To define behavior when the element is present and the value is not null, use the following:

```
<?if:element_name!=?>desired behavior <?end if?>
```

- To define behavior when the element is present, but is null, use the following:

```
<?if:element_name and element_name=""?>desired behavior <?end if?>
```

- To define behavior when the element is not present, use the following:

```
<?if:not(element_name)?>desired behavior <?end if?>
```

## Regrouping the XML Data

The RTF template supports the XSL 2.0 for-each-group standard that allows you to regroup XML data into hierarchies that are not present in the original data. With this feature, your template does not have to follow the hierarchy of the source XML file. You are therefore no longer limited by the structure of your data source.

### XML Sample

To demonstrate the for-each-group standard, the following XML data sample of a CD catalog listing will be regrouped in a template:

```
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD>
    <TITLE>Hide Your Heart</TITLE>
    <ARTIST>Bonnie Tylor</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
  </CD>
  <CD>
    <TITLE>Still got the blues</TITLE>
    <ARTIST>Gary More</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>Virgin Records</COMPANY>
    <PRICE>10.20</PRICE>
    <YEAR>1990</YEAR>
  </CD>
  <CD>
    <TITLE>This is US</TITLE>
    <ARTIST>Gary Lee</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>Virgin Records</COMPANY>
    <PRICE>12.20</PRICE>
    <YEAR>1990</YEAR>
  </CD>
</CATALOG>
```

Using the regrouping syntax, you can create a report of this data that groups the CDs by country and then by year. You are not limited by the data structure presented.

### Regrouping Syntax

To regroup the data, use the following syntax:

```
<?for-each-group: BASE-GROUP; GROUPING-ELEMENT?>
```

For example, to regroup the CD listing by COUNTRY, enter the following in your template:

```
<?for-each-group:CD;COUNTRY?>
```

The elements that were at the same hierarchy level as COUNTRY are now children of COUNTRY. You can then refer to the elements of the group to display the values desired.

To establish nested groupings within the already defined group, use the following syntax:

```
<?for-each:current-group(); GROUPING-ELEMENT?>
```

For example, after declaring the CD grouping by COUNTRY, you can then further group by YEAR within COUNTRY as follows:

```
<?for-each:current-group();YEAR?>
```

At runtime, XML Publisher will loop through the occurrences of the new groupings, displaying the fields that you defined in your template.

**Note:** This syntax is a simplification of the XSL for-each-group syntax. If you choose not to use the simplified syntax above, you can use the XSL syntax as shown below. The XSL syntax can only be used within a form field of the template.

```
<xsl:for-each-group
  select=expression
  group-by="string expression"
  group-adjacent="string expression"
  group-starting-with=pattern>
  <!--Content: (xsl:sort*, content-constructor) -->
</xsl:for-each-group>
```

### Template Example

The following figure shows a template that displays the CDs by Country, then Year, and lists the details for each CD:

Group by Country								
Country:USA								
Group by Year								
Year:2000								
<table border="1"><thead><tr><th>Title</th><th>Artist</th><th>Price</th></tr></thead><tbody><tr><td>Group:DetailsMy CD</td><td>John Doe</td><td>1.00End Group</td></tr></tbody></table>	Title	Artist	Price	Group:DetailsMy CD	John Doe	1.00End Group		
Title	Artist	Price						
Group:DetailsMy CD	John Doe	1.00End Group						
End Group by Year								
End Group by Country								

The following table shows the XML Publisher syntax entries made in the form fields of the preceding template:

Default Text Entry	Form Field Help Text Entry	Description
Group by Country	<code>&lt;?for-each-group:CD;COUNTRY?&gt;</code>	The <code>&lt;?for-each-group:CD;COUNTRY?&gt;</code> tag declares the new group. It regroups the existing CD group by the COUNTRY element.
USA	<code>&lt;?COUNTRY?&gt;</code>	Placeholder to display the data value of the COUNTRY tag.
Group by Year	<code>&lt;?for-each-group:current-group();YEAR?&gt;</code>	The <code>&lt;?for-each-group:current-group();YEAR?&gt;</code> tag regroups the current group (that is, COUNTRY), by the YEAR element.
2000	<code>&lt;?YEAR?&gt;</code>	Placeholder to display the data value of the YEAR tag.
Group: Details	<code>&lt;?for-each:current-group() ?&gt;</code>	Once the data is grouped by COUNTRY and then by YEAR, the <code>&lt;?for-each:current-group() ?&gt;</code> command is used to loop through the elements of the current group (that is, YEAR) and render the data values (TITLE, ARTIST, and PRICE) in the table.
My CD	<code>&lt;?TITLE?&gt;</code>	Placeholder to display the data value of the TITLE tag.
John Doe	<code>&lt;?ARTIST?&gt;</code>	Placeholder to display the data value of the ARTIST tag.
1.00	<code>&lt;?PRICE?&gt;</code>	Placeholder to display the data value of the PRICE tag.
End Group	<code>&lt;?end for-each?&gt;</code>	Closes out the <code>&lt;?for-each:current-group() ?&gt;</code> tag.

Default Text Entry	Form Field Help Text Entry	Description
End Group by Year	<?end for-each-group?>	Closes out the <?for-each-group:current-group();YEAR?> tag.
End Group by Country	<?end for-each-group?>	Closes out the  <?for-each-group:CD;COUNTRY?> tag.

This template produces the following output when merged with the XML file:

Country:USA		
Year:1985		
Title	Artist	Price
Empire Burlesque	Bob Dylan	10.90
Country:UK		
Year:1988		
Title	Artist	Price
Hide your heart	Bonnie Tylor	9.90
Year:1990		
Title	Artist	Price
Still got the blues	Gary More	10.20
This is US	Gary Lee	12.20

### Regrouping by an Expression

Regrouping by an expression allows you to apply a function or command to a data element, and then group the data by the returned result.

To use this feature, state the expression within the regrouping syntax as follows:

```
<?for-each:BASE-GROUP;GROUPING-EXPRESSION?>
```

#### Example

To demonstrate this feature, an XML data sample that simply contains average

temperatures per month will be used as input to a template that calculates the number of months having an average temperature within a certain range.

The following XML sample is composed of <temp> groups. Each <temp> group contains a <month> element and a <degree> element, which contains the average temperature for that month:

```
<temps>
  <temp>
    <month>Jan</month>
    <degree>11</degree>
  </temp>
  <temp>
    <month>Feb</month>
    <degree>14</degree>
  </temp>
  <temp>
    <month>Mar</month>
    <degree>16</degree>
  </temp>
  <temp>
    <month>Apr</month>
    <degree>20</degree>
  </temp>
  <temp>
    <month>May</month>
    <degree>31</degree>
  </temp>
  <temp>
    <month>Jun</month>
    <degree>34</degree>
  </temp>
  <temp>
    <month>Jul</month>
    <degree>39</degree>
  </temp>
  <temp>
    <month>Aug</month>
    <degree>38</degree>
  </temp>
  <temp>
    <month>Sep</month>
    <degree>24</degree>
  </temp>
  <temp>
    <month>Oct</month>
    <degree>28</degree>
  </temp>
  <temp>
    <month>Nov</month>
    <degree>18</degree>
  </temp>
  <temp>
    <month>Dec</month>
    <degree>8</degree>
  </temp>
</temps>
```

You want to display this data in a format showing temperature ranges and a count of the months that have an average temperature to satisfy those ranges, as follows:

## Annual Temperature Summary

Range	Number of Months
0 F to 10 F	1 Month(s)
10 F to 20 F	4 Month(s)
20 F to 30 F	3 Month(s)
30 F to 40 F	4 Month(s)

Using the for-each-group command you can apply an expression to the `<degree>` element that will enable you to group the temperatures by increments of 10 degrees. You can then display a count of the members of each grouping, which will be the number of months having an average temperature that falls within each range.

The template to create the above report is shown in the following figure:

### Annual Temperature Summary

Range	Number of Months
Group by TmpRng	Months Month(s)End TmpRng
Range	

The following table shows the form field entries made in the template:

Default Text Entry	Form Field Help Text Entry
Group by TmpRng	<code>&lt;?for-each-group:temp;floor(degree div 10?)&gt;</code> <code>&lt;?sort:floor(degree div 10)?&gt;</code>
Range	<code>&lt;?concat(floor(degree div 10)*10,' F to ',floor(degree div 10)*10+10, F')?&gt;</code>
Months	<code>&lt;?count(current-group())?&gt;</code>
End TmpRng	<code>&lt;?end for-each-group?&gt;</code>

Note the following about the form field tags:

- The `<?for-each-group:temp;floor(degree div 10)?>` is the regrouping tag. It specifies that for the existing `<temp>` group, the elements are to be regrouped by the expression, `floor(degree div 10)`. The `floor` function is an XSL function that returns the highest integer that is not greater than the argument

(for example, 1.2 returns 1, 0.8 returns 0).

In this case, it returns the value of the `<degree>` element, which is then divided by 10. This will generate the following values from the XML data: 1, 1, 1, 2, 3, 3, 3, 3, 2, 2, 1, and 0.

These are sorted, so that when processed, the following four groups will be created: 0, 1, 2, and 3.

- The `<?concat(floor(degree div 10)*10, 'F to ', floor(degree div 10)*10+10, 'F'?)>` displays the temperature ranges in the row header in increments of 10. The expression concatenates the value of the current group times 10 with the value of the current group times 10 plus 10.

Therefore, for the first group, 0, the row heading displays 0 to (0 +10), or "0 F to 10 F".

- The `<?count(current-group())?)>` uses the count function to count the members of the current group (the number of temperatures that satisfy the range).
- The `<?end for-each-group?)>` tag closes out the grouping.

## Using Variables

Updateable variables differ from standard XSL variables `<xsl:variable>` in that they are updateable during the template application to the XML data. This allows you to create many new features in your templates that require updateable variables.

The variables use a "set and get" approach for assigning, updating, and retrieving values.

Use the following syntax to declare/set a variable value:

```
<?xdoxslt:set_variable($_XDOCTX, 'variable name', value)?>
```

Use the following syntax to retrieve a variable value:

```
<?xdoxslt:get_variable($_XDOCTX, 'variable name')?>
```

You can use this method to perform calculations. For example:

```
<?xdoxslt:set_variable($_XDOCTX, 'x', xdoxslt:get_variable($_XDOCTX, 'x') + 1)?>
```

This sets the value of variable 'x' to its original value plus 1, much like using "x = x + 1".

The `$_XDOCTX` specifies the global document context for the variables. In a multi-threaded environment there may be many transformations occurring at the same time, therefore the variable must be assigned to a single transformation.

See the section on Running Totals, page 2-79 for an example of the usage of updateable variables.



## Defining Parameters

You can pass runtime parameter values into your template. These can then be referenced throughout the template to support many functions. For example, you can filter data in the template, use a value in a conditional formatting block, or pass property values (such as security settings) into the final document.

### Using a parameter in a template

1. Declare the parameter in the template.

Use the following syntax to declare the parameter:

```
<?param@begin:parameter_name;parameter_value?>
```

where

*parameter\_name* is the name of the parameter

*parameter\_value* is the default value for the parameter (the *parameter\_value* is optional)

*param@begin:* is a required string to push the parameter declaration to the top of the template at runtime so that it can be referred to globally in the template.

The syntax must be declared in the Help Text field of a form field. The form field can be placed anywhere in the template.

2. Refer to the parameter in the template by prefixing the name with a "\$" character. For example, if you declare the parameter name to be "InvThresh", then reference the value using "\$InvThresh".

3. At runtime, pass the parameter to the XML Publisher engine programmatically.

Prior to calling the FOProcessor API create a Properties class and assign a property to it for the parameter value as follows:

```
Properties prop = new Properties();  
prop.put("xslt.InvThresh", "1000");
```

### Example: Passing an invoice threshold parameter

This example illustrates how to declare a parameter in your template that will filter your data based on the value of the parameter.

The following XML sample lists invoice data:

```

<INVOICES>
  <INVOICE>
    <INVOICE_NUM>981110</INVOICE_NUM>
    <AMOUNT>1100</AMOUNT>
  </INVOICE>
  <INVOICE>
    <INVOICE_NUM>981111</INVOICE_NUM>
    <AMOUNT>250</AMOUNT>
  </INVOICE>
  <INVOICE>
    <INVOICE_NUM>981112</INVOICE_NUM>
    <AMOUNT>8343</AMOUNT>
  </INVOICE>
  . . .
</INVOICES>

```

The following figure displays a template that accepts a parameter value to limit the invoices displayed in the final document based on the parameter value.

#### InvThresh Declaration

Invoice Number	Invoice Amount
FE IF 13222-2	\$100.00 EI EFE

Field	Form Field Help Text Entry	Description
InvThreshDeclaration	<?param@begin:InvThresh?>	Declares the parameter InvThresh.
FE	<?for-each:INVOICE?>	Begins the repeating group for the INVOICE element.
IF	<?if:AMOUNT>\$InvThresh?>	Tests the value of the AMOUNT element to determine if it is greater than the value of InvThresh.
13222-2	<?INVOICE_NUM?>	Placeholder for the INVOICE_NUM element.
\$100.00	<?AMOUNT?>	Placeholder for the AMOUNT element.
EI	<?end if?>	Closing tag for the if statement.
EFE	<?end for-each?>	Closing tag for the for-each loop.

In this template, only INVOICE elements with an AMOUNT greater than the InvThresh

parameter value will be displayed. If we pass in a parameter value of 1,000, the following output shown in the following figure will result:

Invoice Number	Invoice Amount
981110	1100
981112	8343

Notice the second invoice does not display because its amount was less than the parameter value.

## Setting Properties

XML Publisher properties that are available in the XML Publisher Configuration file can alternatively be embedded into the RTF template. The properties set in the template are resolved at runtime by the XML Publisher engine. You can either hard code the values in the template or embed the values in the incoming XML data. Embedding the properties in the template avoids the use of the configuration file.

**Note:** See XML Publisher Configuration File, page 7-1 for more information about the XML Publisher Configuration file and the available properties.

For example, if you use a nonstandard font in your template, rather than specify the font location in the configuration file, you can embed the font property inside the template. If you need to secure the generated PDF output, you can use the XML Publisher PDF security properties and obtain the password value from the incoming XML data.

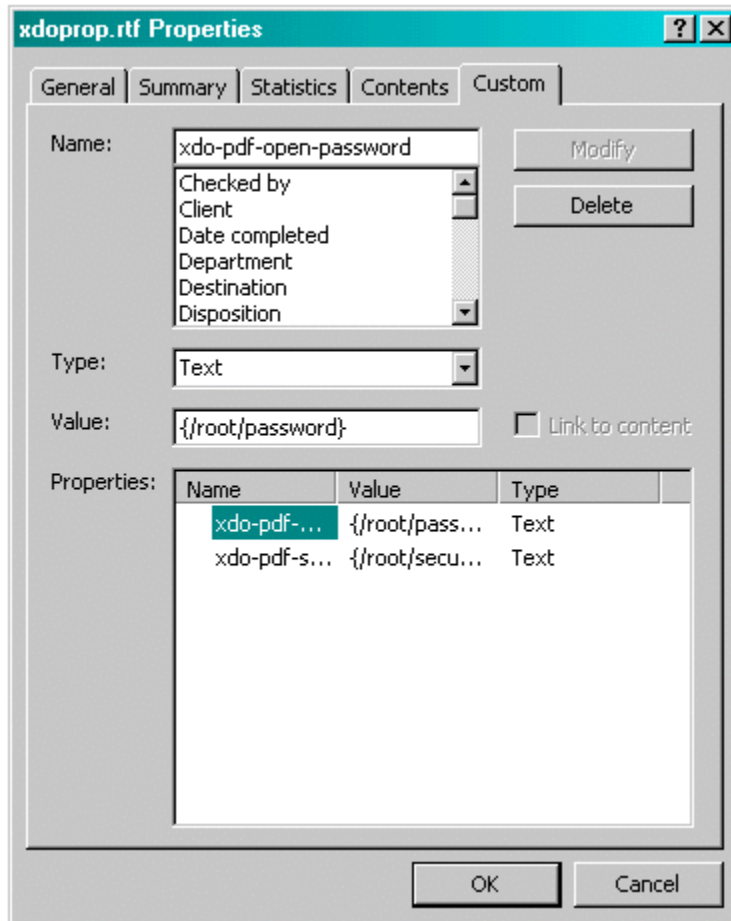
To add an XML Publisher property to a template, use the Microsoft Word **Properties** dialog (available from the **File** menu), and enter the following information:

**Name** - enter the XML Publisher property name prefixed with "xdo-"

**Type** - select "Text"

**Value** - enter the property value. To reference an element from the incoming XML data, enter the path to the XML element enclosed by curly braces. For example:  
{/root/password}

The following figure shows the Properties dialog:



### Embedding a Font Reference

For this example, suppose you want to use a font in the template called "XMLPScript". This font is not available as a regular font on your server, therefore you must tell XML Publisher where to find the font at runtime. You tell XML Publisher where to find the font by setting the "font" property. Assume the font is located in "/tmp/fonts", then you would enter the following in the Properties dialog:

**Name:** xdo-font.XMLPScript.normal.normal

**Type:** Text

**Value:** truetype./tmp/fonts/XMLPScript.ttf

When the template is applied to the XML data on the server, XML Publisher will look for the font in the /tmp/fonts directory. Note that if the template is deployed in multiple locations, you must ensure that the path is valid for each location.

For more information about setting font properties, see Font Definitions, page 7-14.

### Securing a PDF Output

For this example, suppose you want to use a password from the XML data to secure the PDF output document. The XML data is as follows:

```
<PO>
  <security>true</security>
  <password>welcome</password>
  <PO_DETAILS>
    ..
  </PO>
```

In the Properties dialog set two properties: pdf-security to set the security feature as enabled or not, and pdf-open-password to set the password. Enter the following in the Properties dialog:

**Name:** xdo-pdf-security

**Type:** Text

**Value:** {/PO/security}

**Name:** xdo-pdf-open-password

**Type:** Text

**Value:** {/PO/password}

Storing the password in the XML data is not recommended if the XML will persist in the system for any length of time. To avoid this potential security risk, you can use a template parameter value that is generated and passed into the template at runtime.

For example, you could set up the following parameters:

- PDFSec - to pass the value for the xdo-pdf-security property
- PDFPWD - to pass the value for the password

You would then enter the following in the Properties dialog:

**Name:** xdo-pdf-security

**Type:** Text

**Value:** {\$PDFSec}

**Name:** xdo-pdf-open-password

**Type:** Text

**Value:** {\$PDFPWD}

For more information about template parameters, see *Defining Parameters in Your Template*, page 2-89.

## Advanced Report Layouts

### Batch Reports

It is a common requirement to print a batch of documents, such as invoices or purchase orders in a single PDF file. Because these documents are intended for different

customers, each document will require that the page numbering be reset and that page totals are specific to the document. If the header and footer display fields from the data (such as customer name) these will have to be reset as well.

XML Publisher supports this requirement through the use of a context command. This command allows you to define elements of your report to a specific section. When the section changes, these elements are reset.

The following example demonstrates how to reset the header and footer and page numbering within an output file:

The following XML sample is a report that contains multiple invoices:

```
...
<LIST_G_INVOICE>
  <G_INVOICE>
    <BILL_CUST_NAME>Vision, Inc. </BILL_CUST_NAME>
    <TRX_NUMBER>2345678</TRX_NUMBER>
    ...
  </G_INVOICE>
  <G_INVOICE>
    <BILL_CUST_NAME>Oracle, Inc. </BILL_CUST_NAME>
    <TRX_NUMBER>2345685</TRX_NUMBER>
    ...
  </G_INVOICE>
  ...
</LIST_G_INVOICE>
...
```

Each G\_INVOICE element contains an invoice for a potentially different customer. To instruct XML Publisher to start a new section for each occurrence of the G\_INVOICE element, add the @section command to the opening for-each statement for the group, using the following syntax:

```
<?for-each@section:group name?>
```

where *group\_name* is the name of the element for which you want to begin a new section.

For example, the for-each grouping statement for this example will be as follows:

```
<?for-each@section:G_INVOICE?>
```

The closing <?end for-each?> tag is not changed.

The following figure shows a sample template. Note that the G\_INVOICE group for-each declaration is still within the body of the report, even though the headers will be reset by the command.

Invoice# <?TRX\_NUMBER?>

for-each G\_INVOICE

**INVOICE**

BILL\_CUST\_NAME  
BILL\_ADDRESS1  
BILL\_ADDRESS2  
BILL\_CITY, BILL\_STATE BILL\_POSTAL\_CODE

end G\_INVOICE

The following table shows the values of the form fields from the example:

Default Text Entry	Form Field Help Text	Description
for-each G_INVOICE	<?for-each@section:G_INVOICE?>	Begins the G_INVOICE group, and defines the element as a Section. For each occurrence of G_INVOICE, a new section will be started.
<?TRX_NUMBER?>	N/A	Microsoft Word does not support form fields in the header, therefore the placeholder syntax for the TRX_NUMBER element is placed directly in the template.
end G_INVOICE	<?end for-each?>	Closes the G_INVOICE group.

Now for each new occurrence of the G\_INVOICE element, a new section will begin. The page numbers will restart, and if header or footer information is derived from the data, it will be reset as well.

## Cross-Tab Support

The columns of a cross-tab report are data dependent. At design-time you do not know how many columns will be reported, or what the appropriate column headings will be. Moreover, if the columns should break onto a second page, you need to be able to define the row label columns to repeat onto subsequent pages. The following example

shows how to design a simple cross-tab report that supports these features.

This example uses the following XML sample:

```
<ROWSET>
  <RESULTS>
    <INDUSTRY>Motor Vehicle Dealers</INDUSTRY>
    <YEAR>2005</YEAR>
    <QUARTER>Q1</QUARTER>
    <SALES>1000</SALES>
  </RESULTS>
  <RESULTS>
    <INDUSTRY>Motor Vehicle Dealers</INDUSTRY>
    <YEAR>2005</YEAR>
    <QUARTER>Q2</QUARTER>
    <SALES>2000</SALES>
  </RESULTS>
  <RESULTS>
    <INDUSTRY>Motor Vehicle Dealers</INDUSTRY>
    <YEAR>2004</YEAR>
    <QUARTER>Q1</QUARTER>
    <SALES>3000</SALES>
  </RESULTS>
  <RESULTS>
    <INDUSTRY>Motor Vehicle Dealers</INDUSTRY>
    <YEAR>2004</YEAR>
    <QUARTER>Q2</QUARTER>
    <SALES>3000</SALES>
  </RESULTS>
  <RESULTS>
    <INDUSTRY>Motor Vehicle Dealers</INDUSTRY>
    <YEAR>2003</YEAR>
    ...
  </RESULTS>
  <RESULTS>
    <INDUSTRY>Home Furnishings</INDUSTRY>
    ...
  </RESULTS>
  <RESULTS>
    <INDUSTRY>Electronics</INDUSTRY>
    ...
  </RESULTS>
  <RESULTS>
    <INDUSTRY>Food and Beverage</INDUSTRY>
    ...
  </RESULTS>
</ROWSET>
```

From this XML we will generate a report that shows each industry and totals the sales by year as shown in the following figure:



Industry	2005	2004	2003
Motor Vehicle Dealers	3000	6000	1200
Home Furnishings	3200	7770	3300
Electronics	9000	9000	4300
Food and Beverage	1200	900	5600

The template to generate this report is shown in the following figure. The form field entries are shown in the subsequent table.

<b>Industry</b> header column	for: YEAR end
for: INDUSTRY	for: sum(SALES) end end

The form fields in the template have the following values:

Default Text Entry	Form Field Help Text	Description
header column	<?horizontal-break-table:1?>	Defines the first column as a header that should repeat if the table breaks across pages. For more information about this syntax, see Defining Columns to Repeat Across Pages, page 2-99.
for:	<?for-each-group@column:RESULTS;YEAR?>	Uses the regrouping syntax (see Regrouping the XML Data, page 2-82) to group the data by YEAR; and the @column context command to create a table column for each group (YEAR). For more information about context commands, see Using the Context Commands, page 2-123.
YEAR	<?YEAR?>	Placeholder for the YEAR element.
end	<?end for-each-group?>	Closes the for-each-group loop.
for:	<?for-each-group:RESULTS;INDUSTRY?>	Begins the group to create a table row for each INDUSTRY.
INDUSTRY	<?INDUSTRY?>	Placeholder for the INDUSTRY element.

Default Text Entry	Form Field Help Text	Description
for:	<code>&lt;?for-each-group@cell:current-group();YEAR?&gt;</code>	Uses the regrouping syntax (see Regrouping the XML Data, page 2-82) to group the data by YEAR; and the <code>@cell</code> context command to create a table cell for each group (YEAR).
sum(Sales)	<code>&lt;?sum(current-group()//SALES)?&gt;</code>	Sums the sales for the current group (YEAR).
end	<code>&lt;?end for-each-group?&gt;</code>	Closes the for-each-group statement.
end	<code>&lt;?end for-each-group?&gt;</code>	Closes the for-each-group statement.

Note that only the first row uses the `@column` context to determine the number of columns for the table. All remaining rows need to use the `@cell` context to create the table cells for the column. (For more information about context commands, see Using the Context Commands, page 2-123.)

## Dynamic Data Columns

The ability to construct dynamic data columns is a very powerful feature of the RTF template. Using this feature you can design a template that will correctly render a table when the number of columns required by the data is variable.

For example, you are designing a template to display columns of test scores within specific ranges. However, you do not know how many ranges will have data to report. You can define a dynamic data column to split into the correct number of columns at runtime.

Use the following tags to accommodate the dynamic formatting required to render the data correctly:

- Dynamic Column Header

```
<?split-column-header:group element name?>
```

Use this tag to define which group to split for the column headers of a table.

- Dynamic Column `<?split-column-data:group element name?>`

Use this tag to define which group to split for the column data of a table.

- Dynamic Column Width

```
<?split-column-width:name?> or
```

```
<?split-column-width:@width?>
```

Use one of these tags to define the width of the column when the width is described in the XML data. The width can be described in two ways:

- An XML element stores the value of the width. In this case, use the syntax `<?split-column-width: name?>`, where *name* is the XML element tag name that contains the value for the width.
- If the element defined in the `split-column-header` tag, contains a `width` attribute, use the syntax `<?split-column-width:@width?>` to use the value of that attribute.
- Dynamic Column Width's unit value (in points) `<?split-column-width-unit: value?>`

Use this tag to define a multiplier for the column width. If your column widths are defined in character cells, then you will need a multiplier value of ~6 to render the columns to the correct width in points. If the multiplier is not defined, the widths of the columns are calculated as a percentage of the total width of the table. This is illustrated in the following table:

Width Definition	Column 1 (Width = 10)	Column 2 (Width = 12)	Column 3 (Width = 14)
Multiplier not present - % width	10/10+12+14*100 28%	%Width = 33%	%Width =39%
Multiplier = 6 - width	60 pts	72 pts	84 pts

## Defining Columns to Repeat Across Pages

If your table columns expand horizontally across more than one page, you can define how many row heading columns you want to repeat on every page. Use the following syntax to specify the number of columns to repeat:

`<?horizontal-break-table: number?>`

where *number* is the number of columns (starting from the left) to repeat.

Note that this functionality is supported for PDF output only..

## Example of Dynamic Data Columns

A template is required to display test score ranges for school exams. Logically, you want the report to be arranged as shown in the following table:

Test Score	Test Score Range 1	Test Score Range 2	Test Score Range 3	... Test Score Range <i>n</i>
Test Category	# students in Range 1	# students in Range 2	# students in Range 3	# of students in Range <i>n</i>

but you do not know how many Test Score Ranges will be reported. The number of Test Score Range columns is dynamic, depending on the data.

The following XML data describes these test scores. The number of occurrences of the element `<TestScoreRange>` will determine how many columns are required. In this case there are five columns: 0-20, 21-40, 41-60, 61-80, and 81-100. For each column there is an amount element (`<NumOfStudents>`) and a column width attribute (`<TestScore width="15">`).

```
<?xml version="1.0" encoding="utf-8"?>
<TestScoreTable>
  <TestScores>
    <TestCategory>Mathematics</TestCategory>
    <TestScore width="15">
      <TestScoreRange>0-20</TestScoreRange>
      <NumofStudents>30</NumofStudents>
    </TestScore>
    <TestScore width="20">
      <TestScoreRange>21-40</TestScoreRange>
      <NumofStudents>45</NumofStudents>
    </TestScore>
    <TestScore width="15">
      <TestScoreRange>41-60</TestScoreRange>
      <NumofStudents>50</NumofStudents>
    </TestScore>
    <TestScore width="20">
      <TestScoreRange>61-80</TestScoreRange>
      <NumofStudents>102</NumofStudents>
    </TestScore>
    <TestScore width="15">
      <TestScoreRange>81-100</TestScoreRange>
      <NumofStudents>22</NumofStudents>
    </TestScore>
  </TestScores>
</TestScoreTable>
```

Using the dynamic column tags in form fields, set up the table in two columns as shown in the following figure. The first column, "Test Score" is static. The second column, "Column Header and Splitting" is the dynamic column. At runtime this column will split according to the data, and the header for each column will be appropriately populated. The Default Text entry and Form Field Help entry for each field are listed in the table following the figure. (See Form Field Method, page 2-9 for more information on using form fields).

Test Score	Column Header and Splitting
Group:TestScores Test Category	Content and Splitting end:TestScores

Default Text Entry	Form Field Help Text Entry
Group:TestScores	<?for-each:TestScores?>
Test Category	<?TestCategory?>
Column Header and Splitting	<?split-column-header:TestScore?> <?split-column-width:@width?> <?TestScoreRange?>%
Content and Splitting	<?split-column-data:TestScore?> <?NumofStudents?>
end:TestScores	<?end for-each?>

- **Test Score** is the boilerplate column heading.
- Test Category is the placeholder for the <TestCategory> data element, that is, "Mathematics," which will also be the row heading.
- The second column is the one to be split dynamically. The width you specify will be divided by the number of columns of data. In this case, there are 5 data columns.
- The second column will contain the dynamic "range" data. The width of the column will be divided according to the split column width. Because this example does not contain the unit value tag (<?split-column-width-unit:value?>), the column will be split on a percentage basis. Wrapping of the data will occur if required.

**Note:** If the tag (<?split-column-width-unit:value?>) were present, then the columns would have a specific width in points. If the total column widths were wider than the allotted space on the page, then the table would break onto another page.

The "horizontal-break-table" tag could then be used to specify how many columns to repeat on the subsequent page. For example, a value of "1" would repeat the column "Test Score" on the subsequent page, with the continuation of the columns that did not fit on the first page.

The template will render the output shown in the following figure:

Test Score	0-20	21-40	41-60	61-80	81-100
Mathematics	30	45	50	102	22

## Number and Date Formatting

### Number Formatting

XML Publisher supports two methods for specifying the number format:

- Microsoft Word's Native number format mask
- Oracle's format-number function

**Note:** You can also use the native XSL format-number function to format numbers. See: Native XSL Number Formatting, page 2-127.

Use only one of these methods. If the number format mask is specified using both methods, the data will be formatted twice, causing unexpected behavior.

The group separator and the number separator will be set at runtime based on the template locale. This is applicable for both the Oracle format mask and the MS format mask.

### Data Source Requirements

To use the Oracle format mask or the Microsoft format mask, the numbers in your data source must be in a raw format, with no formatting applied (for example: 1000.00). If the number has been formatted for European countries (for example: 1.000,00) the format will not work.

**Note:** The XML Publisher parser requires the Java BigDecimal string representation. This consists of an optional sign ("-") followed by a sequence of zero or more decimal digits (the integer), optionally followed by a fraction, and optionally followed by an exponent. For example: -123456.3455e-3.

### Translation Considerations

If you are designing a template to be translatable, using currency in the Microsoft format mask is not recommended unless you want the data reported in the same currency for all translations. Using the MS format mask sets the currency in the template so that it cannot be updated at runtime.

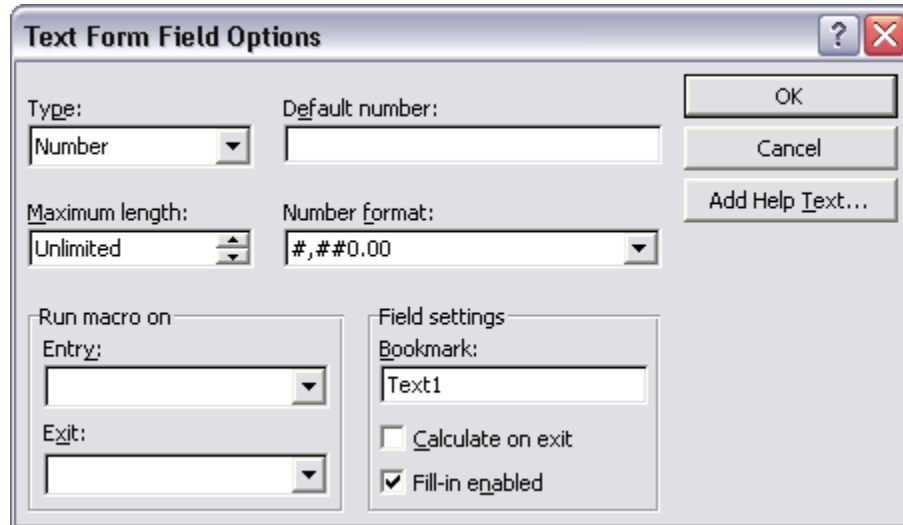
Instead, use the Oracle format mask. For example, L999G999G999D99, where "L" will be

replaced by the currency symbol based on the locale at runtime.

Do not include "%" in the format mask because this will fix the location of the percent sign in the number display, while the desired position could be at the beginning or the end of a number, depending on the locale.

### Using the Microsoft Number Format Mask

To format numeric values, use Microsoft Word's field formatting features available from the Text Form Field Options dialog box. The following graphic displays an example:



To apply a number format to a form field:

1. Open the **Form Field Options** dialog box for the placeholder field.
2. Set the **Type** to Number.
3. Select the appropriate **Number format** from the list of options.

### Supported Microsoft Format Mask Definitions

The following table lists the supported Microsoft format mask definitions:

Symbol	Location	Meaning
0	Number	<p>Digit. Each explicitly set 0 will appear, if no other number occupies the position.</p> <p>Example:</p> <p>Format mask: 00.0000</p> <p>Data: 1.234</p> <p>Display: 01.2340</p>
#	Number	<p>Digit. When set to #, only the incoming data is displayed.</p> <p>Example:</p> <p>Format mask: ##.####</p> <p>Data: 1.234</p> <p>Display: 1.234</p>
.	Number	<p>Determines the position of the decimal separator. The decimal separator symbol used will be determined at runtime based on template locale.</p> <p>For example:</p> <p>Format mask: #,##0.00</p> <p>Data: 1234.56</p> <p>Display for English locale: 1,234.56</p> <p>Display for German locale: 1.234,56</p>
-	Number	<p>Determines placement of minus sign for negative numbers.</p>
,	Number	<p>Determines the placement of the grouping separator. The grouping separator symbol used will be determined at runtime based on template locale.</p> <p>For example:</p> <p>Format mask: #,##0.00</p> <p>Data: 1234.56</p> <p>Display for English locale: 1,234.56</p> <p>Display for German locale: 1.234,56</p>



Symbol	Location	Meaning
E	Number	Separates mantissa and exponent in a scientific notation.  Example:  0.###E+0 plus sign always shown for positive numbers  0.###E-0 plus sign not shown for positive numbers
;	Subpattern boundary	Separates positive and negative subpatterns. See Note below.
%	Prefix or Suffix	Multiply by 100 and show as percentage
'	Prefix or Suffix	Used to quote special characters in a prefix or suffix.

**Note:** Subpattern boundary: A pattern contains a positive and negative subpattern, for example, "#,##0.00;(#,##0.00)". Each subpattern has a prefix, numeric part, and suffix. The negative subpattern is optional. If absent, the positive subpattern prefixed with the localized minus sign ("- " in most locales) is used as the negative subpattern. That is, "0.00" alone is equivalent to "0.00;-0.00". If there is an explicit negative subpattern, it serves only to specify the negative prefix and suffix. The number of digits, minimal digits, and other characteristics are all the same as the positive pattern. That means that "#,##0.0#;(#)" produces precisely the same behavior as "#,##0.0#;(#,##0.0#)".

## Using the Oracle Format Mask

To apply the Oracle format mask to a form field:

1. Open the Form Field Options dialog box for the placeholder field.
2. Set the **Type** to "Regular text".
3. In the Form Field Help Text field, enter the mask definition according to the following example:

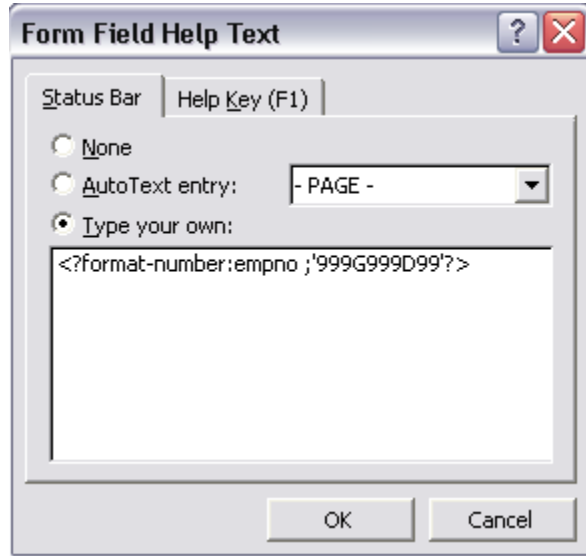
```
<?format-number:fieldname;'999G999D99'??>
```

where

*fieldname* is the XML tag name of the data element you are formatting and

*999G999D99* is the mask definition.

The following graphic shows an example Form Field Help Text dialog entry for the data element "empno":



The following table lists the supported Oracle number format mask symbols and their definitions:

Symbol	Meaning
0	<p>Digit. Each explicitly set 0 will appear, if no other number occupies the position.</p> <p>Example:</p> <p>Format mask: 00.0000</p> <p>Data: 1.234</p> <p>Display: 01.2340</p>
9	<p>Digit. Returns value with the specified number of digits with a leading space if positive or a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.</p> <p>Example:</p> <p>Format mask: 99.9999</p> <p>Data: 1.234</p> <p>Display: 1.234</p>
C	<p>Returns the ISO currency symbol in the specified position.</p>

Symbol	Meaning
D	<p>Determines the placement of the decimal separator. The decimal separator symbol used will be determined at runtime based on template locale.</p> <p>For example:</p> <p>Format mask: 9G999D99</p> <p>Data: 1234.56</p> <p>Display for English locale: 1,234.56</p> <p>Display for German locale: 1.234,56</p>
EEEE	Returns a value in scientific notation.
G	<p>Determines the placement of the grouping (thousands) separator. The grouping separator symbol used will be determined at runtime based on template locale.</p> <p>For example:</p> <p>Format mask: 9G999D99</p> <p>Data: 1234.56</p> <p>Display for English locale: 1,234.56</p> <p>Display for German locale: 1.234,56</p>
L	Returns the local currency symbol in the specified position.
MI	Displays negative value with a trailing "-".
PR	Displays negative value enclosed by <
PT	Displays negative value enclosed by ()
S (before number)	Displays positive value with a leading "+" and negative values with a leading "-"
S (after number)	Displays positive value with a trailing "+" and negative value with a trailing "-"

## Date Formatting

XML Publisher supports three methods for specifying the date format:

- Specify an explicit date format mask using Microsoft Word's native date format mask.

- Specify an explicit date format mask using Oracle's format-date function.
- Specify an abstract date format mask using Oracle's abstract date format masks. (Recommended for multilingual templates.)

Only one method should be used. If both the Oracle and MS format masks are specified, the data will be formatted twice causing unexpected behavior.

### Data Source Requirements

To use the Microsoft format mask or the Oracle format mask, the date from the XML data source must be in canonical format. This format is:

YYYY-MM-DDThh:mm:ss±HH:MM

where

- YYYY is the year
- MM is the month
- DD is the day
- T is the separator between the date and time component
- hh is the hour in 24-hour format
- mm is the minutes
- ss is the seconds
- ±HH:MM is the time zone offset from Universal Time (UTC), or Greenwich Mean Time

An example of this construction is:

2005-01-01T09:30:10-07:00

The data after the "T" is optional, therefore the following date: 2005-01-01 can be formatted using either date formatting option. Note that if you do not include the time zone offset, the time will be formatted to the UTC time.

### Translation Considerations

If you are designing a template to be translatable, explicitly setting a date format mask is not recommended. This is because the date format mask is part of the template, and all published reports based on this template will have the same date format regardless of locale.

For translatable templates, it is recommended that you use the Oracle abstract date format.

If it is necessary to explicitly specify a format mask, the Oracle format mask is

recommended over the MS format mask to ensure future compatibility.

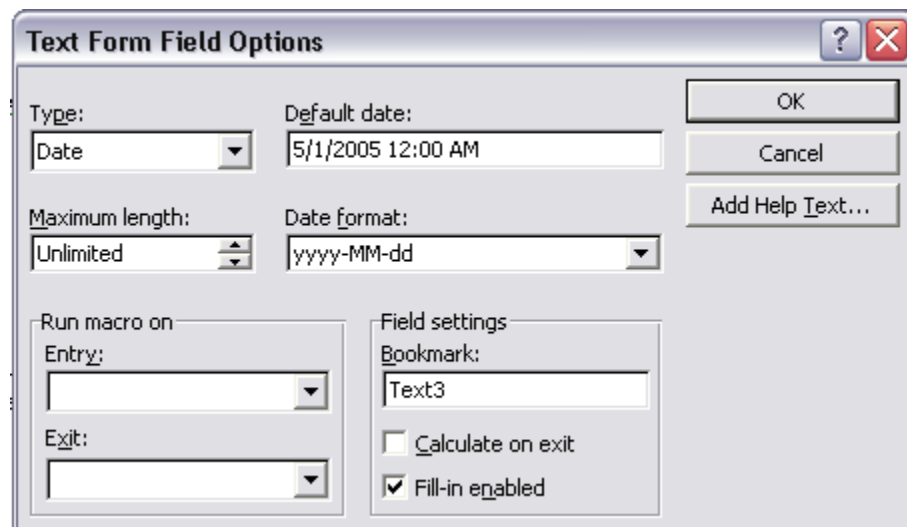
### Using the Microsoft Date Format Mask

To apply a date format to a form field:

1. Open the **Form Field Options** dialog box for the placeholder field.
2. Set the **Type** to Date, Current Date, or Current Time.
3. Select the appropriate **Date format** from the list of options.

If you do not specify the mask in the **Date format** field, the abstract format mask "MEDIUM" will be used as default. See Oracle Abstract Format Masks, page 2-115 for the description.

The following figure shows the Text Form Field Options dialog box with a date format applied:



The following table lists the supported Microsoft date format mask components:

Symbol	Meaning
d	The day of the month. Single-digit days will not have a leading zero.
dd	The day of the month. Single-digit days will have a leading zero.
ddd	The abbreviated name of the day of the week, as defined in AbbreviatedDayNames.
dddd	The full name of the day of the week, as defined in DayNames.

Symbol	Meaning
M	The numeric month. Single-digit months will not have a leading zero.
MM	The numeric month. Single-digit months will have a leading zero.
MMM	The abbreviated name of the month, as defined in <code>AbbreviatedMonthNames</code> .
MMMM	The full name of the month, as defined in <code>MonthNames</code> .
yy	The year without the century. If the year without the century is less than 10, the year is displayed with a leading zero.
yyyy	The year in four digits.
gg	The period or era. This pattern is ignored if the date to be formatted does not have an associated period or era string.
h	The hour in a 12-hour clock. Single-digit hours will not have a leading zero.
hh	The hour in a 12-hour clock. Single-digit hours will have a leading zero.
H	The hour in a 24-hour clock. Single-digit hours will not have a leading zero.
HH	The hour in a 24-hour clock. Single-digit hours will have a leading zero.
m	The minute. Single-digit minutes will not have a leading zero.
mm	The minute. Single-digit minutes will have a leading zero.
s	The second. Single-digit seconds will not have a leading zero.
ss	The second. Single-digit seconds will have a leading zero.
f	Displays seconds fractions represented in one digit.
ff	Displays seconds fractions represented in two digits.
fff	Displays seconds fractions represented in three digits.
ffff	Displays seconds fractions represented in four digits.

Symbol	Meaning
fffff	Displays seconds fractions represented in five digits.
ffffff	Displays seconds fractions represented in six digits.
fffffff	Displays seconds fractions represented in seven digits.
tt	The AM/PM designator defined in AMDesignator or PMDesignator, if any.
z	Displays the time zone offset for the system's current time zone in whole hours only. (This element can be used for formatting only)
zz	Displays the time zone offset for the system's current time zone in whole hours only. (This element can be used for formatting only)
zzz	Displays the time zone offset for the system's current time zone in hours and minutes.
:	The default time separator defined in TimeSeparator.
/	The default date separator defined in DateSeparator.
'	Quoted string. Displays the literal value of any string between two ' characters.
"	Quoted string. Displays the literal value of any string between two " characters.

### Using the Oracle Format Mask

To apply the Oracle format mask to a date field:

1. Open the **Form Field Options** dialog box for the placeholder field.
2. Set the **Type** to Regular Text.
3. Select the **Add Help Text...** button to open the **Form Field Help Text** dialog.
4. Insert the following syntax to specify the date format mask:

```
<?format-date:date_string;
'ABSTRACT_FORMAT_MASK'; 'TIMEZONE'??>
```

or

```
<?format-date-and-calendar:date_string;
'ABSTRACT_FORMAT_MASK'; 'CALENDAR_NAME'; 'TIMEZONE'??>
```

where time zone is optional. The detailed usage of format mask, calendar and time zone is described below.

If no format mask is specified, the abstract format mask "MEDIUM" will be used as default.

Example form field help text entry:

```
<?format-date:hiredate;'YYYY-MM-DD' ?>
```

The following table lists the supported Oracle format mask components:

Symbol	Meaning
- / , . ; : "text"	Punctuation and quoted text are reproduced in the result.
AD A.D.	AD indicator with or without periods.
AM A.M.	Meridian indicator with or without periods.
BC B.C.	BC indicator with or without periods.
CC	Century. For example, 2002 returns 21; 2000 returns 20.
DAY	Name of day, padded with blanks to length of 9 characters.
D	Day of week (1-7).
DD	Day of month (1-31).
DDD	Day of year (1-366).



Symbol	Meaning
DL	Returns a value in the long date format.
DS	Returns a value in the short date format.
DY	Abbreviated name of day.
E	Abbreviated era name.
EE	Full era name.
FF[1..9]	Fractional seconds. Use the numbers 1 to 9 after FF to specify the number of digits in the fractional second portion of the datetime value returned.  Example: 'HH:MI:SS.FF3'
HH	Hour of day (1-12).
HH12	Hour of day (1-12).
HH24	Hour of day (0-23).
MI	Minute (0-59).
MM	Month (01-12; JAN = 01).
MON	Abbreviated name of month.
MONTH	Name of month, padded with blanks to length of 9 characters.
PM	Meridian indicator with or without periods.
P.M.	
RR	Lets you store 20th century dates in the 21st century using only two digits.
RRRR	Round year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you don't want this functionality, then simply enter the 4-digit year.
SS	Seconds (0-59).

Symbol	Meaning
TZD	Daylight savings information. The TZD value is an abbreviated time zone string with daylight savings information. It must correspond to the region specified in TZR.  Example:  PST (for Pacific Standard Time)  PDT (for Pacific Daylight Time)
TZH	Time zone hour. (See TZM format element.)
TZM	Time zone minute. (See TZH format element.)  Example:  'HH:MI:SS.FFTZH:TZM'
TZR	Time zone region information. The value must be one of the time zone regions supported in the database. Example: PST (Pacific Standard Time)
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
X	Local radix character.
YYYY	4-digit year.
YY	Last 2, or 1 digit(s) of year.
Y	

### Default Format Mask

If you do not want to specify a format mask with either the MS method or the Oracle method, you can omit the mask definition and use the default format mask. The default format mask is the MEDIUM abstract format mask from Oracle. (See Oracle Abstract Format Masks, page 2-115 for the definition.)

To use the default option using the Microsoft method, set the **Type** to Date, but leave the **Date format** field blank in the **Text Form Field Options** dialog.

To use the default option using the Oracle method, do not supply a mask definition to the "format-date" function call, for example:

<?format-date:hiredate?>

### Oracle Abstract Format Masks

The abstract date format masks reflect the default implementations of date/time formatting in the I18N library. When you use one of these masks, the output generated will depend on the locale associated with the report.

Specify the abstract mask using the following syntax:

<?format-date:fieldname;'MASK'?>

where fieldname is the XML element tag and

MASK is the Oracle abstract format mask name

For example:

<?format-date:hiredate;'SHORT'?>

<?format-date:hiredate;'LONG\_TIME\_TZ'?>

The following table lists the abstract format masks and the sample output that would be generated for US locale:

Mask	Output for US Locale
SHORT	2/31/99
MEDIUM	Dec 31, 1999
LONG	Friday, December 31, 1999
SHORT_TIME	12/31/99 6:15 PM
MEDIUM_TIME	Dec 31, 1999 6:15 PM
LONG_TIME	Friday, December 31, 1999 6:15 PM
SHORT_TIME_TZ	12/31/99 6:15 PM GMT
MEDIUM_TIME_TZ	Dec 31, 1999 6:15 PM GMT
LONG_TIME_TZ	Friday, December 31, 1999 6:15 PM GMT

# Calendar and Time Zone Support

## Calendar Specification

The term "calendar" refers to the calendar date displayed in the published report. The following types are supported:

- GREGORIAN
- ARABIC\_HIJRAH
- ENGLISH\_HIJRAH
- JAPANESE\_IMPERIAL
- THAI\_BUDDHA
- ROC\_OFFICIAL (Taiwan)

Use one of the following methods to set the calendar type:

- Call the format-date-and-calendar function and declare the calendar type.

For example:

```
<?format-date-and-calendar:hiredate;'LONG_TIME_TZ';'ROC_OFFICIAL';?>
```

The following graphic shows the output generated using this definition with locale set to zh-TW and time zone set to Asia/Taipei:

中華民國88年12月31日 星期五 下午 2:15 台北

- Set the calendar type using the profile option XDO: Calendar Type (XDO\_CALENDAR\_TYPE).

**Note:** The calendar type specified in the template will override the calendar type set in the profile option.

## Time Zone Specification

There are two ways to specify time zone information:

- Call the format-date or format-date-and-calendar function with the Oracle format.
- Set the user profile option Client Timezone (CLIENT\_TIMEZONE\_ID) in Oracle

Applications.

If no time zone is specified, UTC is used.

In the template, the time zone must be specified as a Java time zone string, for example, America/Los Angeles. The following example shows the syntax to enter in the help text field of your template:

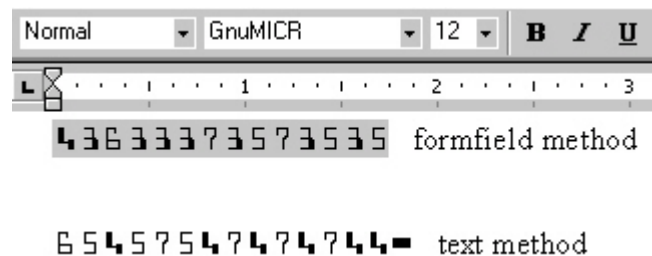
```
<?format-date:hiredate;'LONG_TIME_TZ';'Asia/Shanghai'??>
```

## Using External Fonts

XML Publisher enables you to use fonts in your output that are not normally available on the server. To set up a new font for your report output, use the font to design your template on your client machine, then make it available on the server, and configure XML Publisher to access the font at runtime.

1. Use the font in your template.
  1. Copy the font to your <WINDOWS\_HOME>/fonts directory.
  2. Open Microsoft Word and build your template.
  3. Insert the font in your template: Select the text or form field and then select the desired font from the font dialog box (Format > Font) or font drop down list.

The following graphic shows an example of the form field method and the text method:



2. Place the font on the server.

Place the font in a directory accessible to the formatting engine at runtime.

3. Set the XML Publisher "font" property.

You can set the font property either in the XML Publisher Configuration file or directly in the template.

**To set the property in the configuration file:**

Update the XML Publisher configuration file "fonts" section with the font name and its location on the server. For example, the new entry for a TrueType font is:

```
<font family="MyFontName" style="normal" weight="normal">
  <truetype path="\user\fonts\MyFontName.ttf"/>
</font>
```

See [Setting Runtime Properties](#), page 7-1 for more information.

**To set the property in the template:**

See [Setting Properties](#), page 2-91.

Now you can run your report and XML Publisher will use the font in the output as designed. For PDF output, the advanced font handling features of XML Publisher embed the external font glyphs directly into the final document. The embedded font only contains the glyphs required for the document and not the complete font definition. Therefore the document is completely self-contained, eliminating the need to have external fonts installed on the printer.

## Advanced Barcode Formatting

XML Publisher offers the ability to execute preprocessing on your data prior to applying a barcode font to the data in the output document. For example, you may need to calculate checksum values or start and end bits for the data before formatting them.

The solution requires that you register a barcode encoding class with XML Publisher that can then be instantiated at runtime to carry out the formatting in the template. This is covered in [Advanced Barcode Font Formatting Class Implementation](#), page 8-58.

To enable the formatting feature in your template, you must use two commands in your template. The first command registers the barcode encoding class with XML Publisher. This must be declared somewhere in the template prior to the encoding command. The second is the encoding command to identify the data to be formatted.

### Register the Barcode Encoding Class

Use the following syntax in a form field in your template to register the barcode encoding class:

```
<?register-barcode-vendor:java_class_name;barcode_vendor_id?>
```

This command requires a Java class name (this will carry out the encoding) and a barcode vendor ID as defined by the class. This command must be placed in the template before the commands to encode the data in the template. For example:

```
<?register-barcode-vendor:'oracle.apps.xdo.template.rtf.util.barcodeUtil'.BarcodeUtil;'XMLPBarVendor'?>
```

where

`oracle.apps.xdo.template.rtf.util.barcodeUtil` is the Java class and

`XMLPBarVendor` is the vendor ID that is defined by the class.

## Encode the Data

To format the data, use the following syntax in a form field in your template:

```
<?format-barcode:data;'barcode_type';'barcode_vendor_id'?>
```

where

`data` is the element from your XML data source to be encoded. For example:  
`LABEL_ID`

`barcode_type` is the method in the encoding Java class used to format the data (for example: `Code128a`).

`barcode_vendor_id` is the ID defined in the `register-barcode-vendor` field of the first command you used to register the encoding class.

For example:

```
<?format-barcode:LABEL_ID;'Code128a';'XMLPBarVendor'?>
```

At runtime, the `barcode_type` method is called to format the data value and the barcode font will then be applied to the data in the final output.

## Advanced Design Options

If you have more complex design requirements, XML Publisher supports the use of XSL and XSL:FO elements, and has also extended a set of SQL functions.

RTF templates offer extremely powerful layout options using XML Publisher's syntax. However, because the underlying technology is based on open W3C standards, such as XSL and XPATH, you are not limited by the functionality described in this guide. You can fully utilize the layout and data manipulation features available in these technologies.

## XPath Overview

XPath is an industry standard developed by the World Wide Web Consortium (W3C). It is the method used to navigate through an XML document. XPath is a set of syntax rules for addressing the individual pieces of an XML document. You may not know it, but you have already used XPath; RTF templates use XPath to navigate through the XML data at runtime.

This section contains a brief introduction to XPath principles. For more information, see the W3C Web site: <http://www.w3.org/TR/xpath>

XPath follows the Document Object Model (DOM), which interprets an XML document as a tree of nodes. A node can be one of seven types:

- root
- element

- attribute
- text
- namespace
- processing instruction
- comment

Many of these elements are shown in the following sample XML, which contains a catalog of CDs:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- My CD Listing -->
<CATALOG>
  <CD cattype=Folk>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD cattype=Rock>
    <TITLE>Hide Your Heart</TITLE>
    <ARTIST>Bonnie Tylor</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
  </CD>
</CATALOG>
```

The root node in this example is CATALOG. CD is an element, and it has an attribute cattype. The sample contains the comment My CD Listing. Text is contained within the XML document elements.

## Locating Data

Locate information in an XML document using location-path expressions.

A node is the most common search element you will encounter. Nodes in the example CATALOG XML include CD, TITLE, and ARTIST. Use a path expression to locate nodes within an XML document. For example, the following path returns all CD elements:

```
//CATALOG/CD
```

where

the double slash (//) indicates that all elements in the XML document that match the search criteria are to be returned, regardless of the level within the document.

the slash (/) separates the child nodes. All elements matching the pattern will be returned.

To retrieve the individual TITLE elements, use the following command:

```
/CATALOG/CD/TITLE
```



This example will return the following XML:

```
<CATALOG>
  <CD cattype=Folk>
    <TITLE>Empire Burlesque</TITLE>
  </CD>
  <CD cattype=Rock>
    <TITLE>Hide Your Heart</TITLE>
  </CD>
</CATALOG>
```

Further limit your search by using square brackets. The brackets locate elements with certain child nodes or specified values. For example, the following expression locates all CDs recorded by Bob Dylan:

```
/CATALOG/CD[ARTIST="Bob Dylan"]
```

Or, if each CD element did not have an PRICE element, you could use the following expression to return only those CD elements that include a PRICE element:

```
/CATALOG/CD[PRICE]
```

Use the bracket notation to leverage the attribute value in your search. Use the @ symbol to indicate an attribute. For example, the following expression locates all Rock CDs (all CDs with the cattype attribute value Rock):

```
//CD[@cattype="Rock"]
```

This returns the following data from the sample XML document:

```
<CD cattype=Rock>
  <TITLE>Hide Your Heart</TITLE>
  <ARTIST>Bonnie Tylor</ARTIST>
  <COUNTRY>UK</COUNTRY>
  <PRICE>9.90</PRICE>
  <YEAR>1988</YEAR>
</CD>
```

You can also use brackets to specify the item number to retrieve. For example, the first CD element is read from the XML document using the following XPath expression:

```
/CATALOG/CD[1]
```

The sample returns the first CD element:

```
<CD cattype=Folk>
  <TITLE>Empire Burlesque</TITLE>
  <ARTIST>Bob Dylan</ARTIST>
  <COUNTRY>USA</COUNTRY>
  <PRICE>10.90</PRICE>
  <YEAR>1985</YEAR>
</CD>
```

XPath also supports wildcards to retrieve every element contained within the specified node. For example, to retrieve all the CDs from the sample XML, use the following expression:

```
/CATALOG/*
```

You can combine statements with Boolean operators for more complex searches. The following expression retrieves all Folk and Rock CDs, thus all the elements from the sample:

```
//CD[@catttype="Folk"]||//CD[@catttype="Rock"]
```

The pipe (|) is equal to the logical OR operator. In addition, XPath recognizes the logical OR and AND, as well as the equality operators: <=, <, >, >=, ==, and !=. For example, we can find all CDs released in 1985 or later using the following expression:

```
/CATALOG/CD[YEAR >=1985]
```

### Starting Reference

The first character in an XPath expression determines the point at which it should start in the XML tree. Statements beginning with a forward slash (/) are considered absolute. No slash indicates a relative reference. An example of a relative reference is:

```
CD/*
```

This statement begins the search at the current reference point. That means if the example occurred within a group of statements the reference point left by the previous statement would be utilized.

As noted earlier, double forward slashes (//) retrieve every matching element regardless of location in the document.

### Context and Parent

To select current and parent elements, XPath recognizes the dot notation commonly used to navigate directories. Use a single period (.) to select the current node and use double periods (..) to return the parent of the current node. For example, to retrieve all child nodes of the parent of the current node, use:

```
../*
```

Therefore, to access all CDs from the sample XML, use the following expression:

```
/CATALOG/CD/..
```

You could also access all the CD titles released in 1988 using the following:

```
/CATALOG/CD/TITLE[../YEAR=1988]
```

The .. is used to navigate up the tree of elements to find the YEAR element at the same level as the TITLE, where it is then tested for a match against "1988". You could also use // in this case, but if the element YEAR is used elsewhere in the XML document, you may get erroneous results.

XPath is an extremely powerful standard when combined with RTF templates allowing you to use conditional formatting and filtering in your template.

## Namespace Support

If your XML data contains namespaces, you must declare them in the template prior to referencing the namespace in a placeholder. Declare the namespace in the template using either the basic RTF method or in a form field. Enter the following syntax:

```
<?namespace:namespace name= namespace url?>
```

For example:

```
<?namespace:fsg=http://www.oracle.com/fsg/2002-30-20/?>
```

Once declared, you can use the namespace in the placeholder markup, for example:  
`<?fsg:ReportName?>`

## Using the Context Commands

The XML Publisher syntax is simplified XSL instructions. This syntax, along with any native XSL commands you may use in your template, is converted to XSL-FO when you upload the template to the Template Manager. The placement of these instructions within the converted stylesheet determines the behavior of your template.

XML Publisher's RTF processor places these instructions within the XSL-FO stylesheet according to the most common context. However, sometimes you need to define the context of the instructions differently to create a specific behavior. To support this requirement, XML Publisher provides a set of context commands that allow you to define the context (or placement) of the processing instructions. For example, using context commands, you can:

- Specify an if statement in a table to refer to a cell, a row, a column or the whole table.
- Specify a for-each loop to repeat either the current data or the complete section (to create new headers and footers and restart the page numbering)
- Define a variable in the current loop or at the beginning of the document.

You can specify a context for both processing commands using the XML Publisher syntax and those using native XSL.

- To specify a context for a processing command using the simplified XML Publisher syntax, simply add `@context` to the syntax instruction. For example:
  - `<?for-each@section:INVOICE?>` - specifies that the group INVOICE should begin a new section for each occurrence. By adding the section context, you can reset the header and footer and page numbering.
  - `<?if@column:VAT?>` - specifies that the if statement should apply to the VAT column only.
- To specify a context for an XSL command, add the `xdofo:ctx="context"` attribute to your tags to specify the context for the insertion of the instructions. The value of the context determines where your code is placed.

For example:

```
<xsl:for-each xdofo:ctx="section" select ="INVOICE">
<xsl:attribute xdofo:ctx="inblock"
name="background-color">red</xsl:attribute>
```

XML Publisher supports the following context types:

Context	Description
section	<p>The statement affects the whole section including the header and footer. For example, a <code>for-each@section</code> context command creates a new section for each occurrence - with restarted page numbering and header and footer.</p> <p>See Batch Reports, page 2-93 for an example of this usage.</p>
column	<p>The statement will affect the whole column of a table. This context is typically used to show and hide table columns depending on the data.</p> <p>See Column Formatting, page 2-66 for an example.</p>
cell	<p>The statement will affect the cell of a table. This is often used together with <code>@column</code> in cross-tab tables to create a dynamic number of columns.</p> <p>See Cross-Tab Support, page 2-95 for an example.</p>
block	<p>The statement will affect multiple complete <code>fo:blocks</code> (RTF paragraphs). This context is typically used for <code>if</code> and <code>for-each</code> statements. It can also be used to apply formatting to a paragraph or a table cell.</p> <p>See Cell Highlighting, page 2-71 for an example.</p>
inline	<p>The context will become the single statement inside an <code>fo:inline</code> block. This context is used for variables.</p>
incontext	<p>The statement is inserted immediately after the surrounding statement. This is the default for <code>&lt;?sort?&gt;</code> statements that need to follow the surrounding <code>for-each</code> as the first element.</p>
inblock	<p>The statement becomes a single statement inside an <code>fo:block</code> (RTF paragraph). This is typically not useful for control statements (such as <code>if</code> and <code>for-each</code>) but is useful for statements that generate text, such as <code>call-template</code>.</p>
inlines	<p>The statement will affect multiple complete inline sections. An inline section is text that uses the same formatting, such as a group of words rendered as bold.</p> <p>See If Statements in Boilerplate Text, page 2-63.</p>
begin	<p>The statement will be placed at the beginning of the XSL stylesheet. This is required for global variables. See Defining Parameters, page 2-89.</p>
end	<p>The statement will be placed at the end of the XSL stylesheet.</p>

The following table shows the default context for the XML Publisher commands:

Command	Context
apply-template	inline
attribute	inline
call-template	inblock
choose	block
for-each	block
if	block
import	begin
param	begin
sort	incontext
template	end
value-of	inline
variable	end

## Using XSL Elements

You can use any XSL element in your template by inserting the XSL syntax into a form field.

If you are using the basic RTF method, you cannot insert XSL syntax directly into your template. XML Publisher has extended the following XSL elements for use in RTF templates.

To use these in a basic-method RTF template, you must use the XML Publisher Tag form of the XSL element. If you are using form fields, use either option.

### Apply a Template Rule

Use this element to apply a template rule to the current element's child nodes.

**XSL Syntax:** `<xsl:apply-templates select="name">`

**XML Publisher Tag:** `<?apply:name?>`

This function applies to `<xsl:template-match="n">` where *n* is the element name.

### Copy the Current Node

Use this element to create a copy of the current node.

**XSL Syntax:** `<xsl:copy-of select="name">`

**XML Publisher Tag:** `<?copy-of:name?>`

### Call Template

Use this element to call a named template to be inserted into or applied to the current template. For example, use this feature to render a table multiple times.

**XSL Syntax:** `<xsl:call-template name="name">`

**XML Publisher Tag:** `<?call-template:name?>`

### Template Declaration

Use this element to apply a set of rules when a specified node is matched.

**XSL Syntax:** `<xsl:template name="name">`

**XML Publisher Tag:** `<?template:name?>`

### Variable Declaration

Use this element to declare a local or global variable.

**XSL Syntax:** `<xsl:variable name="name">`

**XML Publisher Tag:** `<?variable:name?>`

**Example:**

```
<xsl:variable name="color" select="'red'"/>
```

Assigns the value "red" to the "color" variable. The variable can then be referenced in the template.

### Import Stylesheet

Use this element to import the contents of one style sheet into another.

**Note:** An imported style sheet has lower precedence than the importing style sheet.

**XSL Syntax:** `<xsl:import href="url">`

**XML Publisher Tag:** `<?import:url?>`

## Define the Root Element of the Stylesheet

This and the `<xsl:stylesheet>` element are completely synonymous elements. Both are used to define the root element of the style sheet.

**Note:** An included style sheet has the same precedence as the including style sheet.

**XSL Syntax:** `<xsl:stylesheet xmlns:x="url">`

**XML Publisher Tag:** `<?namespace:x=url?>`

**Note:** The namespace must be declared in the template. See [Namespace Support](#), page 2-122.

## Native XSL Number Formatting

The native XSL `format-number` function takes the basic format:

`format-number(number, format, [decimalformat])`

Parameter	Description
number	Required. Specifies the number to be formatted.
format	Required. Specifies the format pattern. Use the following characters to specify the pattern: <ul style="list-style-type: none"><li>• # (Denotes a digit. Example: ####)</li><li>• 0 (Denotes leading and following zeros. Example: 0000.00)</li><li>• . (The position of the decimal point Example: ###.##)</li><li>• , (The group separator for thousands. Example: ###,###.##)</li><li>• % (Displays the number as a percentage. Example: ##%)</li><li>• ; (Pattern separator. The first pattern will be used for positive numbers and the second for negative numbers)</li></ul>
decimalformat	Optional. For more information on the decimal format please consult any basic XSLT manual.

## Using FO Elements

You can use the native FO syntax inside the Microsoft Word form fields.

For more information on XSL-FO see the W3C Website at  
<http://www.w3.org/2002/08/XSLFOsummary.html>

The full list of FO elements supported by XML Publisher can be found in the Appendix:  
Supported XSL-FO Elements, page A-1.



---

## Creating a PDF Template

### PDF Template Overview

To create a PDF template, take any existing PDF document and apply the XML Publisher markup. Because the source of the PDF document does not matter, you have multiple design options. For example:

- Design the layout of your template using any application that generates documents that can be converted to PDF
- Scan a paper document to use as a template
- Use a PDF document from a third-party source, such as a Web site

**Note:** The steps required to create a template from a third-party PDF depend on whether form fields have been added to the document. For more information, see *Creating a Template from a Third-Party PDF Form*, page 3-16.

If you are designing the layout, note that once you have converted to PDF, your layout is treated like a set background. When you mark up the template, you draw fields on top of this background. To edit the layout, you must edit your original document and then convert back to PDF.

For this reason, the PDF template is not recommended for documents that will require frequent updates to the layout. However, it is appropriate for forms that will have a fixed layout, such as invoices or purchase orders.

### Supported Modes


XML Publisher supports Adobe Acrobat 5.0 (PDF specification version 1.4). If you are using Adobe Acrobat Professional 6.0 (or later), use the **Reduce File Size Option** (from the **File** menu) to save your file as Adobe Acrobat 5.0 compatible.

For PDF conversion, XML Publisher supports any PDF conversion utility, such as Adobe Acrobat Distiller.

## Designing the Layout

To design the layout of your template you can use any desktop application that generates documents that can be converted to PDF. Or, scan in an original paper document to use as the background for the template.

The following is the layout for a sample purchase order. It was designed using Microsoft Word and converted to PDF using Adobe Acrobat Distiller.

		<b>Purchase Order</b> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">PURCHASE ORDER NO.</td> <td style="width: 25%;">REVISION</td> <td style="width: 25%;">PAGE</td> </tr> </table>		PURCHASE ORDER NO.	REVISION	PAGE									
PURCHASE ORDER NO.	REVISION	PAGE													
<b>VENDOR:</b>		<b>SHIP TO:</b>  													
<b>VENDOR:</b>		<b>BILL TO:</b>  													
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%;">CUSTOMER ACCOUNT NO.</td> <td style="width: 25%;">VENDOR NO.</td> <td style="width: 25%;">DATE OF ORDER/BUYER</td> <td style="width: 25%;">REVISED DATE/BUYER</td> </tr> <tr> <td colspan="2">PAYMENT TERMS</td> <td>SHIP VIA</td> <td>JOB</td> </tr> <tr> <td colspan="2">FREIGHT TERMS</td> <td>REQUESTOR/DELIVER TO</td> <td>CONFIRM TO/TELEPHONE</td> </tr> </table>		CUSTOMER ACCOUNT NO.	VENDOR NO.	DATE OF ORDER/BUYER	REVISED DATE/BUYER	PAYMENT TERMS		SHIP VIA	JOB	FREIGHT TERMS		REQUESTOR/DELIVER TO	CONFIRM TO/TELEPHONE		
CUSTOMER ACCOUNT NO.	VENDOR NO.	DATE OF ORDER/BUYER	REVISED DATE/BUYER												
PAYMENT TERMS		SHIP VIA	JOB												
FREIGHT TERMS		REQUESTOR/DELIVER TO	CONFIRM TO/TELEPHONE												
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="width: 8%;">ITEM</th> <th style="width: 27%;">PART NUMBER/DESCRIPTION</th> <th style="width: 8%;">QUANTITY</th> <th style="width: 8%;">UNIT</th> <th style="width: 10%;">UNIT PRICE</th> <th style="width: 10%;">EXTENSION</th> <th style="width: 8%;">TAX</th> </tr> </table>				ITEM	PART NUMBER/DESCRIPTION	QUANTITY	UNIT	UNIT PRICE	EXTENSION	TAX					
ITEM	PART NUMBER/DESCRIPTION	QUANTITY	UNIT	UNIT PRICE	EXTENSION	TAX									
				<b>Total</b>											
				AUTHORIZED SIGNATURE _____											

Oracle E-Business Suite

The following is the XML data that will be used as input to this template:

```
<?xml version="1.0"?>
<POXPRPOP2>
  <G_HEADERS>
    <POH_PO_NUM>1190-1</POH_PO_NUM>
    <POH_REVISION_NUM>0</POH_REVISION_NUM>
    <POH_SHIP_ADDRESS_LINE1>3455 108th Avenue</POH_SHIP_ADDRESS_LINE1>
    <POH_SHIP_ADDRESS_LINE2></POH_SHIP_ADDRESS_LINE2>
    <POH_SHIP_ADDRESS_LINE3></POH_SHIP_ADDRESS_LINE3>
    <POH_SHIP_ADR_INFO>Seattle, WA 98101</POH_SHIP_ADR_INFO>
    <POH_SHIP_COUNTRY>United States</POH_SHIP_COUNTRY>
    <POH_VENDOR_NAME>Allied Manufacturing</POH_VENDOR_NAME>
    <POH_VENDOR_ADDRESS_LINE1>1145 Brokaw Road</POH_VENDOR_ADDRESS_LINE1>
    <POH_VENDOR_ADR_INFO>San Jose, CA 95034</POH_VENDOR_ADR_INFO>
    <POH_VENDOR_COUNTRY>United States</POH_VENDOR_COUNTRY>
    <POH_BILL_ADDRESS_LINE1>90 Fifth Avenue</POH_BILL_ADDRESS_LINE1>
    <POH_BILL_ADR_INFO>New York, NY 10022-3422</POH_BILL_ADR_INFO>
    <POH_BILL_COUNTRY>United States</POH_BILL_COUNTRY>
    <POH_BUYER>Smith, J</POH_BUYER>
    <POH_PAYMENT_TERMS>45 Net (terms date + 45)</POH_PAYMENT_TERMS>
    <POH_SHIP_VIA>UPS</POH_SHIP_VIA>
    <POH_FREIGHT_TERMS>Due</POH_FREIGHT_TERMS>
    <POH_CURRENCY_CODE>USD</POH_CURRENCY_CODE>
    <POH_CURRENCY_CONVERSION_RATE></POH_CURRENCY_CONVERSION_RATE>
  <LIST_G_LINES>
    <G_LINES>
      <POL_LINE_NUM>1</POL_LINE_NUM>
      <POL_VENDOR_PRODUCT_NUM></POL_VENDOR_PRODUCT_NUM>
      <POL_ITEM_DESCRIPTION>PCMCIA II Card Holder</POL_ITEM_DESCRIPTION>
      <POL_QUANTITY_TO_PRINT></POL_QUANTITY_TO_PRINT>
      <POL_UNIT_OF_MEASURE>Each</POL_UNIT_OF_MEASURE>
      <POL_PRICE_TO_PRINT>15</POL_PRICE_TO_PRINT>
      <C_FLEX_ITEM>CM16374</C_FLEX_ITEM>
      <C_FLEX_ITEM_DISP>CM16374</C_FLEX_ITEM_DISP>
      <PLL_QUANTITY_ORDERED>7500</PLL_QUANTITY_ORDERED>
      <C_AMOUNT_PLL>112500</C_AMOUNT_PLL>
      <C_AMOUNT_PLL_DISP>112,500.00 </C_AMOUNT_PLL_DISP>
    </G_LINES>
  </LIST_G_LINES>
  <C_AMT_POL_RELEASE_TOTAL_ROUND>312420</C_AMT_POL_RELEASE_TOTAL_ROUND>
</G_HEADERS>
</POXPRPOP2>
```

## Adding Markup to the Template Layout

After you have converted your document to PDF, you define form fields that will display the data from the XML input file. These form fields are placeholders for the data.

The process of associating the XML data to the PDF template is the same as the process for the RTF template. See: [Associating the XML Data to the Template Layout: Associating the XML data to the template layout](#), page 2-3.

When you draw the form fields in Adobe Acrobat, you are drawing them *on top* of the layout that you designed. There is not a relationship between the design elements on your template and the form fields. You therefore must place the fields exactly where

you want the data to display on the template

## Creating a Placeholder

You can define a placeholder as text, a check box, or a radio button, depending on how you want the data presented.

**Note:** If you are using Adobe Acrobat 5.0, the **Form Tool** is available from the standard toolbar. If you are using Adobe Acrobat 6.0 or later, display the **Forms Toolbar** from the Tools menu by selecting Tools > Advanced Editing > Forms > Show Forms Toolbar.

### Naming the Placeholder

The name of the placeholder must match the XML source field name.

### Creating a Text Placeholder

To create a text placeholder in your PDF document:

#### Acrobat 5.0 Users:

1. Select the **Form Tool** from the Acrobat toolbar.
2. Draw a form field box in the position on the template where you want the field to display. Drawing the field opens the **Field Properties** dialog box.
3. In the **Name** field of the **Field Properties** dialog box, enter a name for the field.
4. Select **Text** from the **Type** drop down menu.

You can use the **Field Properties** dialog box to set other attributes for the placeholder. For example, enforce maximum character size, set field data type, data type validation, visibility, and formatting.

5. If the field is not placed exactly where desired, drag the field for exact placement.

#### Acrobat 6.0 (and later) Users:

1. Select the **Text Field Tool** from the Forms Toolbar.
2. Draw a form field box in the position on the template where you want the field to display. Drawing the field opens the **Text Field Properties** dialog box.
3. On the **General** tab, enter a name for the placeholder in the **Name** field.

You can use the **Text Field Properties** dialog box to set other attributes for the placeholder. For example, enforce maximum character size, set field data type, data

type validation, visibility, and formatting.

4. If the field is not placed exactly where desired, drag the field for exact placement.

### Supported Field Properties Options

XML Publisher supports the following options available from the **Field Properties** dialog box. For more information about these options, see the Adobe Acrobat documentation.

- **General**
  - Read Only

The setting of this check box in combination with a set of configuration properties control the read-only/updateable state of the field in the output PDF. See Setting Fields as Updateable or Read Only, page 3-15.
- **Appearance**
  - Border Settings: color, background, width, and style
  - Text Settings: color, font, size
  - Common Properties: read only, required, visible/hidden, orientation (in degrees)

(In Acrobat 6.0, these are available from the General tab)
  - Border Style
- **Options** tab
  - Multi-line
  - Scrolling Text
- **Format** tab - Number category options only
- **Calculate** tab - all calculation functions

### Creating a Check Box

A check box is used to present options from which more than one can be selected. Each check box represents a different data element. You define the value that will cause the check box to display as "checked."

For example, a form contains a check box listing of automobile options such as Power Steering, Power Windows, Sunroof, and Alloy Wheels. Each of these represents a different element from the XML file. If the XML file contains a value of "Y" for any of

these fields, you want the check box to display as checked. All or none of these options may be selected.

To create a check box field:

#### **Acrobat 5.0 Users:**

1. Draw the form field.
2. In the **Field Properties** dialog box, enter a **Name** for the field.
3. Select **Check Box** from the **Type** drop down list.
4. Select the **Options** tab.
5. In the **Export Value** field enter the value that the XML data field should match to enable the "checked" state.

For the example, enter "Y" for each check box field.

#### **Acrobat 6.0 (and later) Users:**

1. Select the **Check Box Tool** from the Forms Toolbar.
2. Draw the check box field in the desired position.
3. On the **General** tab of the **Check Box Properties** dialog box, enter a **Name** for the field.
4. Select the **Options** tab.
5. In the **Export Value** field enter the value that the XML data field should match to enable the "checked" state.

For the example, enter "Y" for each check box field.

#### **Creating a Radio Button Group**

A radio button group is used to display options from which only one can be selected.

For example, your XML data file contains a field called <SHIPMENT\_METHOD>. The possible values for this field are "Standard" or "Overnight". You represent this field in your form with two radio buttons, one labeled "Standard" and one labeled "Overnight". Define both radio button fields as placeholders for the <SHIPMENT\_METHOD> data field. For one field, define the "on" state when the value is "Standard". For the other, define the "on" state when the value is "Overnight".

To create a radio button group:

### Acrobat 5.0 Users:

1. Draw the form field.
2. On the **Field Properties** dialog box, enter a **Name** for the field. Each radio button you define to represent this value can be named differently, but must be mapped to the same XML data field.
3. Select **Radio Button** from the **Type** drop down list.
4. Select the **Options** tab.
5. In the **Export Value** field enter the value that the XML data field should match to enable the "on" state.

For the example, enter "Standard" for the field labeled "Standard". Enter "Overnight" for the field labeled "Overnight".

### Acrobat 6.0 (and later) Users:

1. Select the **Radio Button Tool** from the Forms Toolbar.
2. Draw the form field in the position desired on the template.
3. On the **General** tab of the Radio Button Properties dialog, enter a **Name** for the field. Each radio button you define to represent this value can be named differently, but must be mapped to the same XML data field.
4. Select the **Options** tab.
5. In the **Export Value** field enter the value that the XML data field should match to enable the "on" state.

For the example, enter "Standard" for the field labeled "Standard". Enter "Overnight" for the field labeled "Overnight".

## Defining Groups of Repeating Fields

In the PDF template, you explicitly define the area on the page that will contain the repeating fields. For example, on the purchase order template, the repeating fields should display in the block of space between the Item header row and the Total field.

### To define the area to contain the group of repeating fields:

1. Insert a form field at the beginning of the area that is to contain the group. (Acrobat 6.0 users select the **Text Field Tool**, then draw the form field.)
2. In the **Name** field of the **Field Properties** window, enter any unique name you

choose. This field is not mapped.

3. Acrobat 5.0 users: Select **Text** from the **Type** drop down list.
4. In the **Short Description** field (Acrobat 5.0) or the **Tooltip** field (Acrobat 6.0) of the **Field Properties** window, enter the following syntax:  
`<?rep_field="BODY_START"?>`
5. Define the end of the group area by inserting a form field at the end of the area that is to contain the group.
6. In the **Name** field of the **Field Properties** window, enter any unique name you choose. This field is not mapped. Note that the name you assign to this field must be different from the name you assigned to the "body start" field.
7. Acrobat 5.0 users: Select **Text** from the **Type** drop down list.
8. In the **Short Description** field (Acrobat 5.0) or the **Tooltip** field (Acrobat 6.0) of the **Field Properties** window, enter the following syntax:  
`<?rep_field="BODY_END"?>`

#### To define a group of repeating fields:

1. Insert a placeholder for the first element of the group.  
  
**Note:** The placement of this field in relationship to the BODY\_START tag defines the distance between the repeating rows for each occurrence. See Placement of Repeating Fields, page 3-14.
2. For each element in the group, enter the following syntax in the **Short Description** field (Acrobat 5.0) or the **Tooltip** field (Acrobat 6.0):

`<?rep_field="T1_Gn"?>`

where n is the row number of the item on the template.

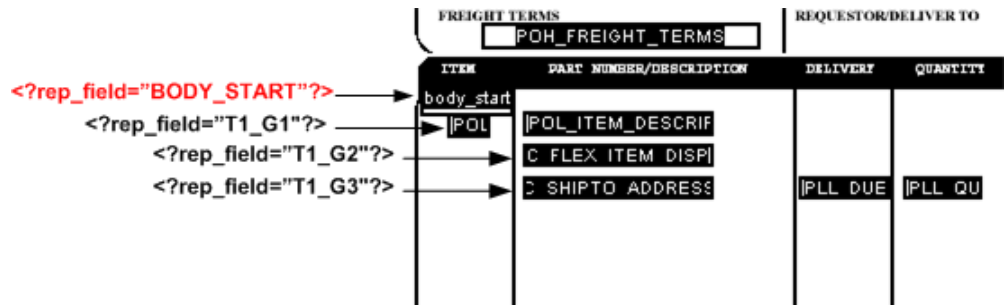
For example, the group in the sample report is laid out in three rows.

- For the fields belonging to the row that begins with "PO\_LINE\_NUM" enter  
`<?rep_field="T1_G1"?>`
- For the fields belonging to the row that begins with "C\_FLEX\_ITEM\_DISP" enter  
`<?rep_field="T1_G2"?>`
- For the fields belonging to the row that begins with "C\_SHIP\_TO\_ADDRESS" enter



<?rep\_field="T1\_G3"?>

The following graphic shows the entries for the **Short Description/Tooltip** field:



3. (Optional) Align your fields. To ensure proper alignment of a row of fields, it is recommended that you use Adobe Acrobat's alignment feature.

## Adding Page Numbers and Page Breaks

This section describes how to add the following page-features to your PDF template:

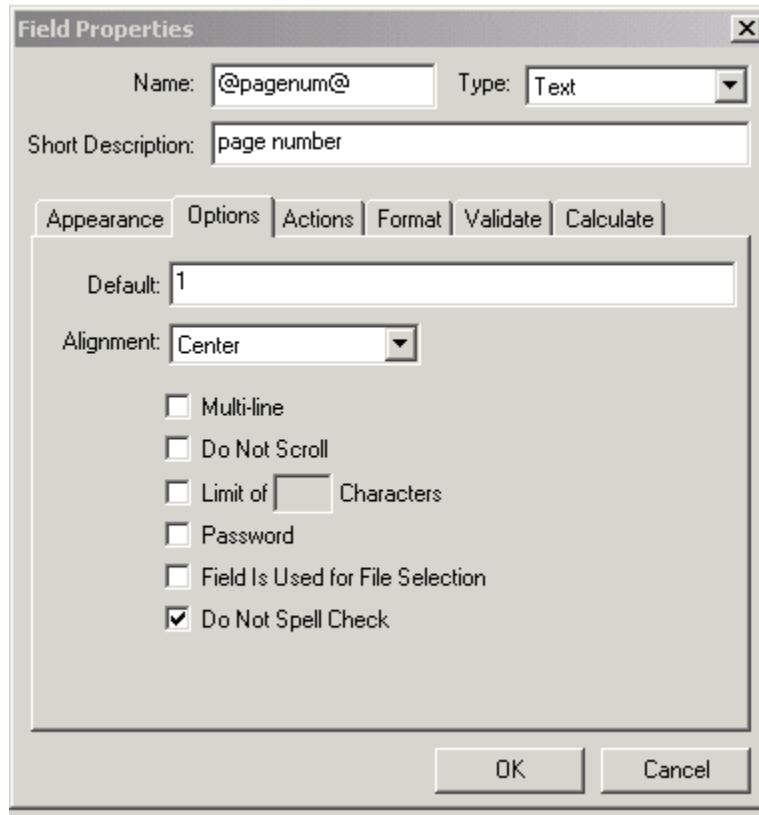
- Page Numbers
- Page Breaks

### Adding Page Numbers

To add page numbers, define a field in the template where you want the page number to appear and enter an initial value in that field as follows:

1. Decide the position on the template where you want the page number to be displayed.
2. Create a placeholder field called @pagenum@ (see Creating a Text Placeholder, page 3-4).
3. Enter a starting value for the page number in the **Default** field. If the XML data includes a value for this field, the start value assigned in the template will be overridden. If no start value is assigned, it will default to 1.

The figure below shows the Field Properties dialog for a page number field:



## Adding Page Breaks

You can define a page break in your template to occur after a repeatable field. To insert a page break after the occurrence of a specific field, add the following to the syntax in the **Short Description** field of the Field Properties dialog box (use the **Tooltip** field for Acrobat 6.0):

```
page_break="yes"
```

For example:

```
<?rep_field="T1_G3", page_break="yes"?>
```

The following example demonstrates inserting a page break in a template. The XML sample contains salaries of employees by department:

```

<?xml version="1.0"?>
<!-- Generated by Oracle Reports version 6.0.8.22.0 -->
<ROOT>
  <LIST_G_DEPTNO>
    <G_DEPTNO>
      <DEPTNO>10</DEPTNO>
      <LIST_G_EMPNO>
        <G_EMPNO>
          <EMPNO>7782</EMPNO>
          <ENAME>CLARK</ENAME>
          <JOB>MANAGER</JOB>
          <SAL>2450</SAL>
        </G_EMPNO>
        <G_EMPNO>
          <EMPNO>7839</EMPNO>
          <ENAME>KING</ENAME>
          <JOB>PRESIDENT</JOB>
          <SAL>5000</SAL>
        </G_EMPNO>
        <G_EMPNO>
          <EMPNO>125</EMPNO>
          <ENAME>KANG</ENAME>
          <JOB>CLERK</JOB>
          <SAL>2000</SAL>
        </G_EMPNO>
        <G_EMPNO>
          <EMPNO>7934</EMPNO>
          <ENAME>MILLER</ENAME>
          <JOB>CLERK</JOB>
          <SAL>1300</SAL>
        </G_EMPNO>
        <G_EMPNO>
          <EMPNO>123</EMPNO>
          <ENAME>MARY</ENAME>
          <JOB>CLERK</JOB>
          <SAL>400</SAL>
        </G_EMPNO>
        <G_EMPNO>
          <EMPNO>124</EMPNO>
          <ENAME>TOM</ENAME>
          <JOB>CLERK</JOB>
          <SAL>3000</SAL>
        </G_EMPNO>
      </LIST_G_EMPNO>
      <SUMSALPERDEPTNO>9150</SUMSALPERDEPTNO>
    </G_DEPTNO>

    <G_DEPTNO>
      <DEPTNO>30</DEPTNO>
      <LIST_G_EMPNO>
        .
        .
        .

      </LIST_G_EMPNO>
      <SUMSALPERDEPTNO>9400</SUMSALPERDEPTNO>
    </G_DEPTNO>
  </LIST_G_DEPTNO>
  <SUMSALPERREPORT>29425</SUMSALPERREPORT>
</ROOT>

```

We want to report the salary information for each employee by department as shown in the following template:

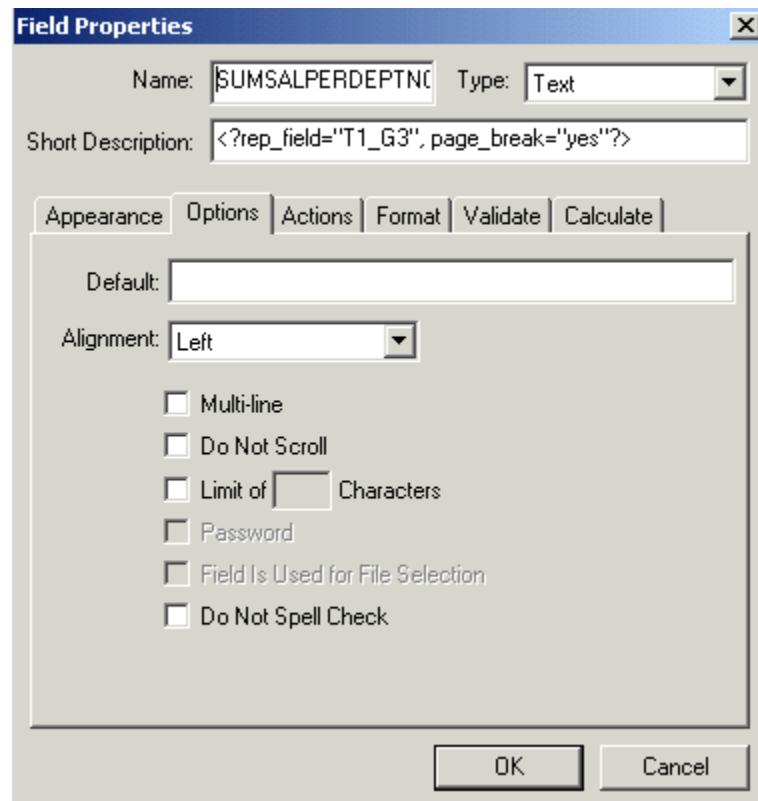
## Department Salary Summary

<b>body_start</b>	Dept No.	Emp No	Emp Name	Job	Salary
	<b>DEPTNO</b>	<b>EMPNO</b>	<b>ENAME</b>	<b>JOB</b>	<b>SAL</b>
					<b>SUMSALPERDEPTNO</b>

To insert a page break after each department, insert the page break syntax in the Short Description (or Tooltip field) for the SUMSALPERDEPTNO field as follows:

```
<?rep_field="T1_G3", page_break="yes"?>
```

The Field Properties dialog box for the field is shown in the following figure:



Note that in order for the break to occur, the field must be populated with data from the XML file.

The sample report with data is shown in the following figure:

Department Salary Summary				
Dept No.	Emp No	Emp Name	Job	Salary
20	7369	SMITH	CLERK	800
	7876	ADAMS	CLERK	1100
	7902	FORD	ANALYST	3000
	7788	SCOTT	ANALYST	3000
	7566	JONES	MANAGER	2975
				10875


## Performing Calculations

Adobe Acrobat provides a calculation function in the **Field Properties** dialog box. To create a field to display a calculated total on your report:

1. Create a text field to display the calculated total. Give the field any **Name** you choose.
2. In the **Field Properties** dialog box, select the **Format** tab.
3. Select **Number** from the **Category** list.
4. Select the **Calculate** tab.
5. Select the radio button next to "Value is the *operation* of the following fields:"
6. Select **sum** from the drop down list.
7. Select the **Pick...** button and select the fields that you want totaled.

## Completed PDF Template

The following figure shows the completed PDF template:



**Purchase Order**  

PURCHASE ORDER NO.	REVISION	PAGE
IPOH PO 01	IPOH	

**SHIP TO:**  

IPOH SHIP ADDRESS
IPOH SHIP ADR INFO1
IPOH SHIP COUNTRY1

**BILL TO:**  

IPOH BILL ADDRESS
IPOH BILL ADR INFO1
IPOH BILL COUNTRY1

**VENDOR:**

IPOH VENDOR NAME
IPOH VENDOR ADDRESS 11
IPOH VENDOR ADR INFO1
IPOH VENDOR COUNTRY

CUSTOMER/ACCOUNT NO.	VENDOR NO.	DATE OF ORDER/BUYER		REVEAL DATE/BUYER
IPOH CUST	IPOH VE	IPOH CR	IPOH ARI	
PAYMENT TERMS		SHIP VIA		FOB
POH PAYMENT TERMS		IPOH SHIP V		<input type="checkbox"/> POH FOB
FREIGHT TERMS		REQUESTOR/DELIVER TO		CONTACT TELEPHONE
POH FREIGHT TERMS				

ITEM	PART NUMBER/DESCRIPTION	DELIVERY	QUANTITY	UNIT	UNIT PRICE	EXTENSION	TAX
body_el IPO	IPOL ITEM DES IC FLEX ITEM IC SHIPTO ADR	IPIT DI	IPIT F	POI	IPOL	IC AMN	IPIT
body_r							

				<b>Total</b>	IC AMOUNT P
				AUTHORIZED SIGNATURE	

Oracle E-Business Suite

## Runtime Behavior

### Placement of Repeating Fields

As already noted, the placement, spacing, and alignment of fields that you create on the template are independent of the underlying form layout. At runtime, XML Publisher places each repeating row of data according to calculations performed on the placement of the rows of fields that you created, as follows:

**First occurrence:**

The first row of repeating fields will display exactly where you have placed them on the template.

**Second occurrence, single row:**

To place the second occurrence of the group, XML Publisher calculates the distance between the BODY\_START tag and the first field of the first occurrence. The first field of the second occurrence of the group will be placed this calculated distance below the first occurrence.

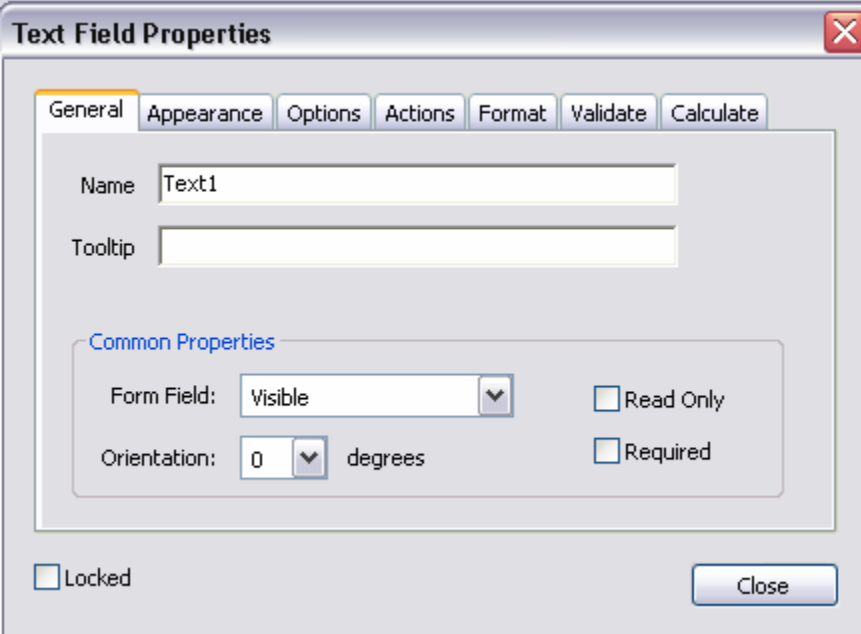
**Second occurrence, multiple rows:**

If the first group contains multiple rows, the second occurrence of the group will be placed the calculated distance below the last row of the first occurrence.

The distance between the rows within the group will be maintained as defined in the first occurrence.

## Setting Fields as Updateable or Read Only

When you define a field in the template you have the option of selecting "Read Only" for the field, as shown in the following sample Text Field Properties dialog:

The image shows a 'Text Field Properties' dialog box with a title bar and a close button. It has several tabs: 'General' (selected), 'Appearance', 'Options', 'Actions', 'Format', 'Validate', and 'Calculate'. In the 'General' tab, there are input fields for 'Name' (containing 'Text1') and 'Tooltip'. Below these is a section titled 'Common Properties' containing a 'Form Field' dropdown menu set to 'Visible', an 'Orientation' dropdown set to '0' degrees, and two checkboxes: 'Read Only' and 'Required', both of which are currently unchecked. At the bottom left is a 'Locked' checkbox, also unchecked, and at the bottom right is a 'Close' button.

Regardless of what you choose at design time for the Read Only check box, the default behavior of the PDF processing engine is to set all fields to read-only for the output PDF. You can change this behavior using the following configuration properties in the XML Publisher Configuration File, page 7-1:

- all-field-readonly

- all-fields-readonly-asis
- remove-pdf-fields

Note that in the first two options, you are setting a state for the field in the PDF output. The setting of individual fields can still be changed in the output using Adobe Acrobat Professional. Also note that because the fields are maintained, the data is still separate and can be extracted. In the third option, "remove-pdf-fields" the structure is flattened and no field/data separation is maintained.

**To make all fields updateable:**

Set the "all-field-readonly" property to "false". This sets the Read Only state to "false" for all fields regardless of the individual field settings at design time.

**To make all fields read only:**

This is the default behavior. No settings are required.

**To maintain the Read Only check box selection for each field:**

To maintain the setting of the Read Only check box on a field-by-field basis in the output PDF, set the property "all-fields-readonly-asis" to "true". This property will override the settings of "all-field-readonly".

**To remove all fields from the output PDF:**

Set the property "remove-pdf-fields" to "true".

## Overflow Data

When multiple pages are required to accommodate the occurrences of repeating rows of data, each page will display identically except for the defined repeating area, which will display the continuation of the repeating data. For example, if the item rows of the purchase order extend past the area defined on the template, succeeding pages will display all data from the purchase order form with the continuation of the item rows.

## Creating a Template from a Third-Party PDF

There are many PDF forms available online that you may want to use as templates for your report data. For example, government forms that your company is required to submit. You can use these downloaded PDF files as your report templates, supplying the XML data at runtime to fill the report out.

Some of these forms already have form fields defined, some do not. If the form fields are not already defined in the downloaded PDF, you must create them. See Adding Markup to the Template Layout, page 3-3 for instructions on inserting the form field placeholders.



## To Use a Third-Party PDF Form as a Template

1. Copy the PDF file to your local system.
2. Open the file in Adobe Acrobat.
3. Select the **Text Field Tool** (Acrobat 6.0 users) or the **Form Tool** (Acrobat 5.0 users). This will highlight text fields that have already been defined.

The following figure shows a sample W-4 PDF form after selecting the **Text Field Tool** to highlight the text fields (in Acrobat 6.0).

The screenshot shows the Adobe Acrobat Professional interface with a W-4 form open. The 'Text Field Tool' is selected in the toolbar, and several text fields on the form are highlighted with black boxes. The form is titled 'W-4 Employee's Withholding Allowance Certificate' and includes fields for personal information, social security number, and employer details. The highlighted fields include the first name and middle initial, last name, social security number, home address, city or town, state, and ZIP code, and the total number of allowances. The form also includes a section for claiming exemption from withholding and a signature line.

To map the existing form fields to the data from your incoming XML file, you must rename the fields to match the element names in your XML file.

4. Open the text form field **Properties** dialog by either double-clicking the field, or by selecting the field then selecting **Properties** from the right-mouse menu.
5. In the **Name** field, enter the element name from your input XML file.
6. Repeat for all fields that you want populated by your data file.



---

## Creating an eText Template

This chapter covers the following topics:

- Introduction
- Outbound eText Templates

### Introduction

An eText template is an RTF-based template that is used to generate text output for Electronic Funds Transfer (EFT) and Electronic Data Interchange (EDI). At runtime, XML Publisher applies this template to an input XML data file to create an output text file that can be transmitted to a bank or other customer. Because the output is intended for electronic communication, the eText templates must follow very specific format instructions for exact placement of data.

**Note:** An EFT is an electronic transmission of financial data and payments to banks in a specific fixed-position format flat file (text).

EDI is similar to EFT except it is not only limited to the transmission of payment information to banks. It is often used as a method of exchanging business documents, such as purchase orders and invoices, between companies. EDI data is delimiter-based, and also transmitted as a flat file (text).

Files in these formats are transmitted as flat files, rather than printed on paper. The length of a record is often several hundred characters and therefore difficult to layout on standard size paper.

To accommodate the record length, the EFT and EDI templates are designed using tables. Each record is represented by a table. Each row in a table corresponds to a field in a record. The columns of the table specify the position, length, and value of the field.

These formats can also require special handling of the data from the input XML file. This special handling can be on a global level (for example, character replacement and

sequencing) or on a record level (for example, sorting). Commands to perform these functions are declared in command rows. Global level commands are declared in setup tables.

At runtime, XML Publisher constructs the output file according to the setup commands and layout specifications in the tables.

## **Prerequisites**

This section is intended for users who are familiar with EDI and EFT transactions. Audience for this section: preparers of eText templates will require both functional and technical knowledge. That is, functional expertise to understand bank and country specific payment format requirements and sufficient technical expertise to understand XML data structure and eText specific coding syntax commands, functions, and operations.

# **Outbound eText Templates**

## **Structure of eText Templates**

There are two types of eText templates: fixed-position based (EFT templates) and delimiter-based (EDI templates). The templates are composed of a series of tables. The tables define layout and setup commands and data field definitions. The required data description columns for the two types of templates vary, but the commands and functions available are the same. A table can contain just commands, or it can contain commands and data fields.

The following graphic shows a sample from an EFT template to display the general structure of command and data rows:

**Format Setup:**

*Hint: Define formatting options...*

<TEMPLATE TYPE>	FIXED POSITION BASED
<OUTPUT CHARACTER SET>	iso-8859-1
<NEW RECORD CHARACTER>	Carriage Return

**Sequences:**

*Hint: Define sequence generators...*

<DEFINE SEQUENCE>	PaymentsSeq
<RESET AT LEVEL>	PayerInstrument
<INCREMENT BASIS>	LEVEL
<END DEFINE SEQUENCE >	PaymentsSeq

} Commands

**Format Data Records:**

<LEVEL>	<POSITION>	<LENGTH>	<FORMAT>	<PAD>	<DATA>	<COMMENTS>
<NEW RECORD>						FileHeaderRec
1	1		Number		1	Record Type Code
2		2	Alpha	R, ' '	'01'	Priority Code
4		1	Alpha	R, ' '		Immediate Blank
5		9	Alpha	R, ' '	BankAccount/BankNumber	Immediate
14		1	Alpha	R, ' '	'1'	Mutually Agreed
15		9	Alpha	R, ' '	Payer/TaxIdentifier	Immediate Origin
24		6	Date		SYSDATE	File Creation Date

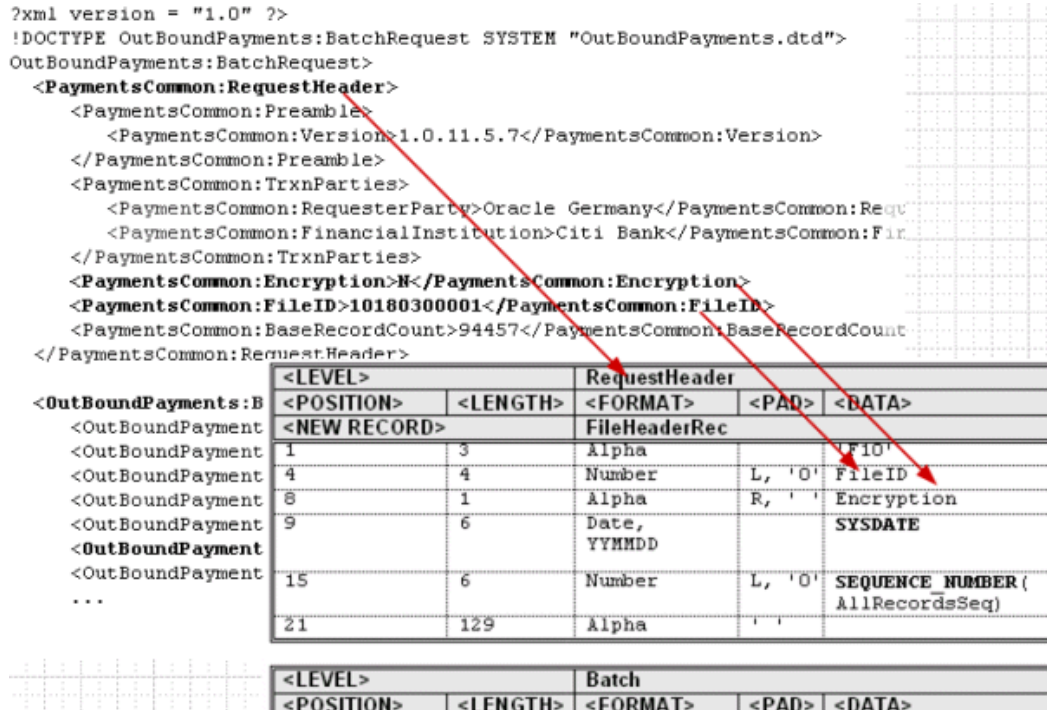
} Data Elements or Functions

Commands that apply globally, or commands that define program elements for the template, are "setup" commands. These must be specified in the initial table(s) of the template. Examples of setup commands are Template Type and Character Set.

In the data tables you provide the source XML data element name (or static data) and the specific placement and formatting definitions required by the receiving bank or entity. You can also define functions to be performed on the data and conditional statements.

The data tables must always start with a command row that defines the "Level." The Level associates the table to an element from the XML data file, and establishes the hierarchy. The data fields that are then defined in the table for the Level correspond to the child elements of the XML element.

The graphic below illustrates the relationship between the XML data hierarchy and the template Level. The XML element "RequestHeader" is defined as the Level. The data elements defined in the table ("FileID" and "Encryption") are children of the RequestHeader element.



The order of the tables in the template determines the print order of the records. At runtime the system loops through all the instances of the XML element corresponding to a table (Level) and prints the records belonging to the table. The system then moves on to the next table in the template. If tables are nested, the system will generate the nested records of the child tables before moving on to the next parent instance.

#### Command Rows, Data Rows, and Data Column Header Rows

The following figure shows the placement of Command Rows, Data Rows, and Data Column Header Rows:

Hint: Define formatting options...

Sequences :

<DEFINE SEQUENCE>	PaymentsSeq
<RESET AT LEVEL>	PayerInstrument
<INCREMENT BASIS>	LEVEL
<END DEFINE SEQUENCE >	PaymentsSeq

A diagram illustrating the relationship between spreadsheet components and database concepts:

- Command Rows**: Points to the **Data Field Column Header**.
- Data Field Column Header**: Points to the **Data Rows**.
- Data Rows**: Points to the **Data Field Column Header**.

<LEVEL>		PayerInstrument				
<POSITION>	<LENGTH>	<FORMAT>	<PAD>	<DATA>		<COMMENT>
<NEW RECORD>		FileHeaderRec				
1	1	Number		1		Record Type
2	2	Alpha	R, ' '	'01'		Priority Code
4	1	Alpha	R, ' '			Immediate D blank
5	9	Alpha	R, ' '	BankAccount/BankNum ber		Immediate D
14	1	Alpha	R, ' '	'1'		Mutually Agree
15	9	Alpha	R, ' '	Payer/TaxIdentifier		Immediate O
24	6	Date,		SYSDATE		File Creation

Blank rows can be inserted anywhere in a table to improve readability. Most often they are used in the setup table, between commands. Blank rows are ignored by XML Publisher when the template is parsed.

Data column headers specify the column headings for the data fields (such as Position, Length, Format, Padding, and Comments). A column header row usually follows the Level command in a table (or the sorting command, if one is used). The column header row must come before any data rows in the table. Additional empty column header rows can be inserted at any position in a table to improve readability. The empty rows will be ignored at runtime.

Data rows contain the data fields to correspond to the column header rows.

Creating an eText Template 4-5

## Constructing the Data Tables

The data tables contain a combination of command rows and data field rows. Each data table must begin with a Level command row that specifies its XML element. Each record must begin with a New Record command that specifies the start of a new record, and the end of a previous record (if any).

The required columns for the data fields vary depending on the Template Type.

### Command Rows

The command rows always have two columns: command name and command parameter. The supported commands are:

- Level
- New record
- Sort ascending
- Sort descending
- Display condition

The usage for each of these commands is described in the following sections.

#### Level Command

The level command associates a table with an XML element. The parameter for the level command is an XML element. The level will be printed once for each instance the XML element appears in the data input file.

The level commands define the hierarchy of the template. For example, Payment XML data extracts are hierarchical. A batch can have multiple child payments, and a payment can have multiple child invoices. This hierarchy is represented in XML as nested child elements within a parent element. By associating the tables with XML elements through the level command, the tables will also have the same hierarchical structure.

Similar to the closing tag of an XML element, the level command has a companion end-level command. The child tables must be defined between the level and end-level commands of the table defined for the parent element.

An XML element can be associated with only one level. All the records belonging to a level must reside in the table of that level or within a nested table belonging to that level. The end-level command will be specified at the end of the final table.

Following is a sample structure of an EFT file record layout:

- FileHeaderRecordA
  - BatchHeaderRecordA



- BatchHeaderRecordB  
PaymentRecordA  
PaymentRecordB
  - InvoiceRecordA
- Batch FooterRecordC
- BatchFooterRecordD
- FileFooterRecordB

Following would be its table layout:

<LEVEL>	<b>RequestHeader</b>
<NEW RECORD>	FileHeaderRecordA
Data rows for the FileHeaderRecordA	
<LEVEL>	<b>Batch</b>
<NEW RECORD>	BatchHeaderRecordA
Data rows for the BatchHeaderRecordA	
<NEW RECORD>	BatchHeaderRecordB
Data rows for the BatchHeaderRecordB	
<LEVEL>	<b>Payment</b>
<NEW RECORD>	PaymentRecordA
Data rows for the PaymentRecordA	
<NEW RECORD>	PaymentRecordB

---

Data rows for the PaymentRecordB

---

---

<LEVEL> **Invoice**

<NEW RECORD> InvoiceRecordA

Data rows for the InvoiceRecordA

<END LEVEL> **Invoice**

---

---

<END LEVEL> **Payment**

---

---

<LEVEL> **Batch**

<NEW RECORD> BatchFooterRecordC

Data rows for the BatchFooterRecordC

<NEW RECORD> BatchFooterRecordD

Data rows for the BatchFooterRecordD

<END LEVEL> **Batch**

---

---

<LEVEL> **RequestHeader**

<NEW RECORD> FileFooterRecordB

Data rows for the FileFooterRecordB

<END LEVEL> **RequestHeader**

---

Multiple records for the same level can exist in the same table. However, each table can only have one level defined. In the example above, the BatchHeaderRecordA and BatchHeaderRecordB are both defined in the same table. However, note that the END

LEVEL for the Payment must be defined in its own separate table after the child element Invoice. The Payment END LEVEL cannot reside in the same table as the Invoice Level.

Note that you do not have to use all the levels from the data extract in your template. For example, if an extract contains the levels: RequestHeader > Batch > Payment > Invoice, you can use just the batch and invoice levels. However, the hierarchy of the levels must be maintained.

The table hierarchy determines the order that the records are printed. For each parent XML element, the records of the corresponding parent table are printed in the order they appear in the table. The system loops through the instances of the child XML elements corresponding to the child tables and prints the child records according to their specified order. The system then prints the records of the enclosing (end-level) parent table, if any.

For example, given the EFT template structure above, assume the input data file contains the following:

- Batch1
  - Payment1
    - Invoice1
    - Invoice2
  - Payment2
    - Invoice1
- Batch2
  - Payment1
    - Invoice1
    - Invoice2
    - Invoice3

This will generate the following printed records:

Record Order	Record Type	Description
1	FileHeaderRecordA	One header record for the EFT file

Record Order	Record Type	Description
2	BatchHeaderRecordA	For Batch1
3	BatchHeaderRecordB	For Batch1
4	PaymentRecordA	For Batch1, Payment1
5	PaymentRecordB	For Batch1, Payment1
6	InvoiceRecordA	For Batch1, Payment1, Invoice1
7	InvoiceRecordA	For Batch1, Payment1, Invoice2
8	PaymentRecordA	For Batch1, Payment2
9	PaymentRecordB	For Batch1, Payment2
10	InvoiceRecordA	For Batch1, Payment2, Invoice1
11	BatchFooterRecordC	For Batch1
12	BatchFooterRecordD	For Batch1
13	BatchHeaderRecordA	For Batch2
14	BatchHeaderRecordB	For Batch2
15	PaymentRecordA	For Batch2, Payment1
16	PaymentRecordB	For Batch2, Payment1
17	InvoiceRecordA	For Batch2, Payment1, Invoice1
18	InvoiceRecordA	For Batch2, Payment1, Invoice2
19	InvoiceRecordA	For Batch2, Payment1, Invoice3

Record Order	Record Type	Description
20	BatchFooterRecordC	For Batch2
21	BatchFooterRecordD	For Batch2
22	FileFooterRecordB	One footer record for the EFT file

### **New Record Command**

The new record command signifies the start of a record and the end of the previous one, if any. Every record in a template must start with the new record command. The record continues until the next new record command, or until the end of the table or the end of the level command.

A record is a construct for the organization of the elements belonging to a level. The record name is not associated with the XML input file.

A table can contain multiple records, and therefore multiple new record commands. All the records in a table are at the same hierarchy level. They will be printed in the order in which they are specified in the table.

The new record command can have a name as its parameter. This name becomes the name for the record. The record name is also referred to as the record type. The name can be used in the COUNT function for counting the generated instances of the record. See COUNT, page 4-28 function, for more information.

Consecutive new record commands (or empty records) are not allowed.

### **Sort Ascending and Sort Descending Commands**

Use the sort ascending and sort descending commands to sort the instances of a level. Enter the elements you wish to sort by in a comma-separated list. This is an optional command. When used, it must come right after the (first) level command and it applies to all records of the level, even if the records are specified in multiple tables.

### **Display Condition Command**

The display condition command specifies when the enclosed record or data field group should be displayed. The command parameter is a boolean expression. When it evaluates to true, the record or data field group is displayed. Otherwise the record or data field group is skipped.

The display condition command can be used with either a record or a group of data fields. When used with a record, the display condition command must follow the new record command. When used with a group of data fields, the display condition command must follow a data field row. In this case, the display condition will apply to the rest of the fields through the end of the record.

Consecutive display condition commands are merged as AND conditions. The merged display conditions apply to the same enclosed record or data field group.

## Structure of the Data Rows

The output record data fields are represented in the template by table rows. In FIXED\_POSITION\_BASED templates, each row has the following attributes (or columns):

- Position
- Length
- Format
- Pad
- Data
- Comments

The first five columns are required and must appear in the order listed.

For DELIMITER\_BASED templates, each data row has the following attributes (columns):

- Maximum Length
- Format
- Data
- Tag
- Comments

The first three columns are required and must be declared in the order stated.

In both template types, the Comments column is optional and ignored by the system. You can insert additional information columns if you wish, as all columns after the required ones are ignored.

The usage rules for these columns are as follows:

### Position

This column is only used with FIXED\_POSITION\_BASED templates. The Position column is a "Comments" column that can be used to indicate the starting position of the field in the record. Use your entries in this column to track the length of your template. The actual positions of fields in the output eText file are determined by the lengths specified in the Length field. See **Length/Maximum Length** below.

Note that although the information in the column is not required, the column is required; furthermore it is a good design practice to note the starting positions.

**Length/Maximum Length**

Specifies the length of the field. The unit is in number of characters. For FIXED\_POSITION\_BASED templates, all the fields are fixed length. If the data is less than the specified length, it is padded. If the data is longer, it is truncated. The truncation always occurs on the right.

For DELIMITER\_BASED templates, the maximum length of the field is specified. If the data exceeds the maximum length, it will be truncated. Data is not padded if it is less than the maximum length.

**Format Column**

Specifies the data type and format setting. There are three accepted data types:

- Alpha
- Number
- Date

Refer to Field Level Key Words, page 4-33 for their usage.

**Number Data Type**

Numeric data has three optional format settings: Integer, Decimal, or you can define a format mask. Specify the optional settings with the Number data type as follows:

- Number, Integer
- Number, Decimal
- Number, <format mask>

For example:

Number, ###,###.00

The Integer format uses only the whole number portion of a numeric value and discards the decimal. The Decimal format uses only the decimal portion of the numeric value and discards the integer portion.

The following table shows examples of how to set a format mask. When specifying the mask, # represents that a digit is to be displayed when present in the data; 0 represents that the digit placeholder is to be displayed whether data is present or not.

When specifying the format mask, the group separator must always be "," and the decimal separator must always be "." To alter these in the actual output, you must use the Setup Commands NUMBER THOUSANDS SEPARATOR and NUMBER DECIMAL SEPARATOR. See Setup Command Tables, page 4-16 for details on these commands.

The following table shows sample Data, Format Specifier, and Output. The Output assumes the default group and decimal separators.

<b>Data</b>	<b>Format Specifier</b>	<b>Output</b>
123456789	###,###.00	123,456,789.00
123456789.2	###.00	123456789.20
1234.56789	###.000	1234.568
123456789.2	#	123456789
123456789.2	###	123456789.2
123456789	###	123456789

#### **Date Data Type**

The Date data type format setting must always be explicitly stated. The format setting follows the SQL date styles, such as MMDDYY.

#### **Mapping EDI Delimiter-Based Data Types to eText Data Types**

Some EDI (DELIMITER\_BASED) formats use more descriptive data types. These are mapped to the three template data types in the following table:

<b>ASC X12 Data Type</b>	<b>Format Template Data Type</b>
A - Alphabetic	Alpha
AN -Alphanumeric	Alpha
B - Binary	Number
CD - Composite data element	N/A
CH - Character	Alpha
DT - Date	Date
FS - Fixed-length string	Alpha
ID - Identifier	Alpha
IV - Incrementing Value	Number



ASC X12 Data Type	Format Template Data Type
Nn - Numeric	Number
PW - Password	Alpha
R - Decimal number	Numer
TM - Time	Date

Now assume you have specified the following setup commands:

NUMBER THOUSANDS SEPARATOR	.
NUMBER DECIMAL SEPARATOR	,

The following table shows the Data, Format Specifier, and Output for this case. Note that the Format Specifier requires the use of the default separators, regardless of the setup command entries.

Data	Format Specifier	Output
123456789	###,###.00	123.456.789,00
123456789.2	###.00	123456789,20
1234.56789	###.000	1234,568
123456789.2	#	123456789
123456789.2	#.##	123456789,2
123456789	###	123456789

### Pad

This applies to FIXED\_POSITION\_BASED templates only. Specify the padding side (L = left or R = right) and the character. Both numeric and alphanumeric fields can be padded. If this field is not specified, Numeric fields are left-padded with "0"; Alpha fields are right-padded with spaces.

Example usage:

- To pad a field on the left with a "0", enter the following in the Pad column field:  
L, '0'
- To pad a field on the right with a space, enter the following the Pad column field:  
R, ''

#### **Data**

Specifies the XML element from the data extract that is to populate the field. The data column can simply contain the XML tag name, or it can contain expressions and functions. For more information, see *Expressions, Control Structure, and Functions*, page 4-27.

#### **Tag**

Acts as a comment column for DELIMITER\_BASED templates. It specifies the reference tag in EDIFACT formats, and the reference IDs in ASC X12.

#### **Comments**

Use this column to note any free form comments to the template. Usually this column is used to note the business requirement and usage of the data field.

## **Setup Command Tables**

### **Setup Command Table**

A template always begins with a table that specifies the setup commands. The setup commands define global attributes, such as template type and output character set and program elements, such as sequencing and concatenation.

The setup commands are:

- Template Type
- Output Character Set
- New Record Character
- Invalid Characters
- Replace Characters
- Number Thousands Separator
- Number Decimal Separator
- Define Level
- Define Sequence
- Define Concatenation

Some example setup tables are shown in the following figures:

XDO file name:  
XINT-01.rtf

*Mapping of Payment Format:*  
**International Payments EFT Format**

**Format Setup:**

*Hint: Define formatting options...*

<TEMPLATE TYPE>	FIXED_POSITION_BASED
<OUTPUT CHARACTER SET>	iso-8859-1
<NEW RECORD CHARACTER>	Carriage Return

<INVALID CHARACTERS>	¿
<REPLACE CHARACTERS>	
A	AO
E	EO
I	IO
O	OO
U	UO
<END REPLACE CHARACTERS>	

**Format Data Levels:**

*Hint: Define data levels that are needed in the format which do not exist in data extract...*

<DEFINE LEVEL>	PaymentsByPayDatePayee
<BASE LEVEL>	Payment
<GROUPING CRITERIA>	'PaymentDate, PayeeName'
<END DEFINE LEVEL>	PaymentsByPayDatePayee

<DEFINE LEVEL>	InvoicesByReportingCatAndAttrib
<BASE LEVEL>	Invoice

<GROUPING CRITERIA>	<pre> 'InvoiceTrxnReportingCat', (IF InvoiceTrxnReportingCat = 'V' THEN   'InvoiceDEVTransitGoods' END IF), (IF InvoiceTrxnReportingCat = 'V' THEN   'InvoiceDEVGoodsIndexNum' END IF), (IF InvoiceTrxnReportingCat = 'V' THEN   'InvoiceDEVPassingTrade' END IF) </pre>
<END DEFINE LEVEL>	InvoicesByReportingCatAndAttrib

### Sequences :

*Hint: Define sequence generators...*

<DEFINE SEQUENCE>	AllRecordsSeq
<RESET AT LEVEL>	PERIODIC_SEQUENCE
<INCREMENT BASIS>	RECORD
<START AT>	BaseRecordCount + 1
<MAXIMUM>	999999
<END DEFINE SEQUENCE >	AllRecordsSeq

<DEFINE SEQUENCE>	PaymentsSeq
<RESET AT LEVEL>	Batch
<INCREMENT BASIS>	LEVEL
<END DEFINE SEQUENCE >	PaymentsSeq

### Concatenated Records :

*Hint: Define fields that are composed of concatenated records...*

<DEFINE CONCATENATION>	ConcatenatedInvoiceInfo
<BASE LEVEL>	Invoice
<ELEMENT>	InvoiceNum
<DELIMITER>	','
<END DEFINE CONCATENATION>	ConcatenatedInvoiceInfo

### **Template Type Command**

This command specifies the type of template. There are two types: FIXED\_POSITION\_BASED and DELIMITER\_BASED.

Use the FIXED\_POSITION\_BASED templates for fixed-length record formats, such as EFTs. In these formats, all fields in a record are a fixed length. If data is shorter than the specified length, it will be padded. If longer, it will be truncated. The system specifies the default behavior for data padding and truncation. Examples of fixed position based formats are EFTs in Europe, and NACHA ACH file in the U.S.

In a DELIMITER\_BASED template, data is never padded and only truncated when it has reached a maximum field length. Empty fields are allowed (when the data is null). Designated delimiters are used to separate the data fields. If a field is empty, two delimiters will appear next to each other. Examples of delimited-based templates are EDI formats such as ASC X12 820 and UN EDIFACT formats - PAYMUL, DIRDEB, and CREMUL.

In EDI formats, a record is sometimes referred to as a segment. An EDI segment is treated the same as a record. Start each segment with a new record command and give

it a record name. You should have a data field specifying the segment name as part of the output data immediately following the new record command.

For DELIMITER\_BASED templates, you insert the appropriate data field delimiters in separate rows between the data fields. After every data field row, you insert a delimiter row. You can insert a placeholder for an empty field by defining two consecutive delimiter rows.

Empty fields are often used for syntax reasons: you must insert placeholders for empty fields so that the fields that follow can be properly identified.

There are different delimiters to signify data fields, composite data fields, and end of record. Some formats allow you to choose the delimiter characters. In all cases you should use the same delimiter consistently for the same purpose to avoid syntax errors.

In DELIMITER\_BASED templates, the <POSITION> and <PAD> columns do not apply. They are omitted from the data tables.

Some DELIMITER\_BASED templates have minimum and maximum length specifications. In those cases Oracle Payments validates the length.

#### **Define Level Command**

Some formats require specific additional data levels that are not in the data extract. For example, some formats require that payments be grouped by payment date. Using the Define Level command, a payment date group can be defined and referenced as a level in the template, even though it is not in the input extract file.

When you use the Define Level command you declare a base level that exists in the extract. The Define Level command inserts a new level one level higher than the base level of the extract. The new level functions as a grouping of the instances of the base level.

The Define Level command is a setup command, therefore it must be defined in the setup table. It has three subcommands:

- Base Level Command - defines the level (XML element) from the extract that the new level is based on. The Define Level command must always have one and only one base level subcommand.
- Grouping Criteria - defines the XML extract elements that are used to group the instances of the base level to form the instances of the new level. The parameter of the grouping criteria command is a comma-separated list of elements that specify the grouping conditions.

The order of the elements determines the hierarchy of the grouping. The instances of the base level are first divided into groups according to the values of the first criterion, then each of these groups is subdivided into groups according to the second criterion, and so on. Each of the final subgroups will be considered as an instance of the new level.

- Group Sort Ascending or Group Sort Descending - defines the sorting of the group. Insert the <GROUP SORT ASCENDING> or <GROUP SORT DESCENDING>

command row anywhere between the <DEFINE LEVEL> and <END DEFINE LEVEL> commands. The parameter of the sort command is a comma-separated list of elements by which to sort the group.

For example, the following table shows five payments under a batch:

<b>Payment Instance</b>	<b>PaymentDate (grouping criterion 1)</b>	<b>PayeeName (grouping criterion 2)</b>
Payment1	PaymentDate1	PayeeName1
Payment2	PaymentDate2	PayeeName1
Payment3	PaymentDate1	PayeeName2
Payment4	PaymentDate1	PayeeName1
Payment5	PaymentDate1	PayeeName3

In the template, construct the setup table as follows to create a level called "PaymentsByPayDatePayee" from the base level "Payment" grouped according to PaymentDate and Payee Name. Add the Group Sort Ascending command to sort ea:

<DEFINE LEVEL>	PaymentsByPayDatePayee
<BASE LEVEL>	Payment
<GROUPING CRITERIA>	PaymentDate, PayeeName
<GROUP SORT ASCENDING>	PaymentDate, PayeeName
<END DEFINE LEVEL>	PaymentsByPayDatePayee

The five payments will generate the following four groups (instances) for the new level:

<b>Payment Group Instance</b>	<b>Group Criteria</b>	<b>Payments in Group</b>
Group1	PaymentDate1, PayeeName1	Payment1, Payment4
Group2	PaymentDate1, PayeeName2	Payment3

Payment Group Instance	Group Criteria	Payments in Group
Group3	PaymentDate1, PayeeName3	Payment5
Group4	PaymentDate2, PayeeName1	Payment2

The order of the new instances is the order that the records will print. When evaluating the multiple grouping criteria to form the instances of the new level, the criteria can be thought of as forming a hierarchy. The first criterion is at the top of the hierarchy, the last criterion is at the bottom of the hierarchy.

Generally there are two kinds of format-specific data grouping scenarios in EFT formats. Some formats print the group records only; others print the groups with the individual element records nested inside groups. Following are two examples for these scenarios based on the five payments and grouping conditions previously illustrated.

### Example

First Scenario: Group Records Only

EFT File Structure:

- BatchRec
  - PaymentGroupHeaderRec
  - PaymentGroupFooterRec

Record Sequence	Record Type	Description
1	BatchRec	
2	PaymentGroupHeaderRec	For group 1 (PaymentDate1, PayeeName1)
3	PaymentGroupFooterRec	For group 1 (PaymentDate1, PayeeName1)
4	PaymentGroupHeaderRec	For group 2 (PaymentDate1, PayeeName2)
5	PaymentGroupFooterRec	For group 2 (PaymentDate1, PayeeName2)
6	PaymentGroupHeaderRec	For group 3 (PaymentDate1, PayeeName3)
7	PaymentGroupFooterRec	For group 3 (PaymentDate1, PayeeName3)
8	PaymentGroupHeaderRec	For group 4 (PaymentDate2, PayeeName1)

Record Sequence	Record Type	Description
9	PaymentGroupFooterRec	For group 4 (PaymentDate2, PayeeName1)

### Example

Scenario 2: Group Records and Individual Records

EFT File Structure:

BatchRec

- PaymentGroupHeaderRec
  - PaymentRec
- PaymentGroupFooterRec

Generated output:

Record Sequence	Record Type	Description
1	BatchRec	
2	PaymentGroupHeaderRec	For group 1 (PaymentDate1, PayeeName1)
3	PaymentRec	For Payment1
4	PaymentRec	For Payment4
5	PaymentGroupFooterRec	For group 1 (PaymentDate1, PayeeName1)
6	PaymentGroupHeaderRec	For group 2 (PaymentDate1, PayeeName2)
7	PaymentRec	For Payment3
8	PaymentGroupFooterRec	For group 2 (PaymentDate1, PayeeName2)
9	PaymentGroupHeaderRec	For group 3 (PaymentDate1, PayeeName3)
10	PaymentRec	For Payment5
11	PaymentGroupFooterRec	For group 3 (PaymentDate1, PayeeName3)



Record Sequence	Record Type	Description
12	PaymentGroupHeaderRec	For group 4 (PaymentDate2, PayeeName1)
13	PaymentRec	For Payment2
14	PaymentGroupFooterRec	For group 4 (PaymentDate2, PayeeName1)

Once defined with the Define Level command, the new level can be used in the template in the same manner as a level occurring in the extract. However, the records of the new level can only reference the base level fields that are defined in its grouping criteria. They cannot reference other base level fields other than in summary functions.

For example, the PaymentGroupHeaderRec can reference the PaymentDate and PayeeName in its fields. It can also reference thePaymentAmount (a payment level field) in a SUM function. However, it cannot reference other payment level fields, such as PaymentDocName or PaymentDocNum.

The Define Level command must always have one and only one grouping criteria subcommand. The Define Level command has a companion end-define level command. The subcommands must be specified between the define level and end-define level commands. They can be declared in any order.

#### **Define Sequence Command**

The define sequence command define a sequence that can be used in conjunction with the SEQUENCE\_NUMBER function to index either the generated EFT records or the extract instances (the database records). The EFT records are the physical records defined in the template. The database records are the records from the extract. To avoid confusion, the term "record" will always refer to the EFT record. The database record will be referred to as an extract element instance or level.

The define sequence command has four subcommands: reset at level, increment basis, start at, and maximum:

#### **Reset at Level**

The reset at level subcommand defines where the sequence resets its starting number. It is a mandatory subcommand. For example, to number the payments in a batch, define the reset at level as Batch. To continue numbering across batches, define the reset level as RequestHeader.

In some cases the sequence is reset outside the template. For example, a periodic sequence may be defined to reset by date. In these cases, the PERIODIC\_SEQUENCE keyword is used for the reset at level. The system saves the last sequence number used for a payment file to the database. Outside events control resetting the sequence in the database. For the next payment file run, the sequence number is extracted from the database for the start at number (see start at subcommand).

**Increment Basis**

The increment basis subcommand specifies if the sequence should be incremented based on record or extract instances. The allowed parameters for this subcommand are RECORD and LEVEL.

Enter RECORD to increment the sequence for every record.

Enter LEVEL to increment the sequence for every new instance of a level.

Note that for levels with multiple records, if you use the level-based increment all the records in the level will have the same sequence number. The record-based increment will assign each record in the level a new sequence number.

For level-based increments, the sequence number can be used in the fields of one level only. For example, suppose an extract has a hierarchy of batch > payment > invoice and you define the increment basis by level sequence, with reset at the batch level. You can use the sequence in either the payment or invoice level fields, but not both. You cannot have sequential numbering across hierarchical levels.

However, this rule does not apply to increment basis by record sequences. Records can be sequenced across levels.

For both increment basis by level and by record sequences, the level of the sequence is implicit based on where the sequence is defined.

**Define Concatenation Command**

Use the define concatenation command to concatenate child-level extract elements for use in parent-level fields. For example, use this command to concatenate invoice number and due date for all the invoices belonging to a payment for use in a payment-level field.

The define concatenation command has three subcommands: base level, element, and delimiter.

**Base Level Subcommand**

The base level subcommand specifies the child level for the operation. For each parent-level instance, the concatenation operation loops through the child-level instances to generate the concatenated string.

**Item Subcommand**

The item subcommand specifies the operation used to generate each item. An item is a child-level expression that will be concatenated together to generate the concatenation string.

**Delimiter Subcommand**

The delimiter subcommand specifies the delimiter to separate the concatenated items in the string.

**Using the SUBSTR Function**

Use the SUBSTR function to break down concatenated strings into smaller strings that can be placed into different fields. For example, the following table shows five invoices in a payment:

Invoice	InvoiceNum
1	car_parts_inv0001
2	car_parts_inv0002
3	car_parts_inv0003
4	car_parts_inv0004
5	car_parts_inv0005

Using the following concatenation definition:

<DEFINE CONCATENATION>	ConcatenatedInvoiceInfo
<BASE LEVEL>	Invoice
<ELEMENT>	InvoiceNum
<DELIMITER>	','
<END DEFINE CONCATENATION>	ConcatenatedInvoiceInfo

You can reference ConcatenatedInvoiceInfo in a payment level field. The string will be:

car\_parts\_inv0001,car\_parts\_inv0002,car\_parts\_inv0003,car\_parts\_inv0004,car\_parts\_inv0005

If you want to use only the first forty characters of the concatenated invoice info, use either TRUNCATE function or the SUBSTR function as follows:

TRUNCATE (ConcatenatedInvoiceInfo, 40)

SUBSTR (ConctenatedInvoiceInfo, 1, 40)

Either of these statements will result in:

car\_parts\_inv0001,car\_parts\_inv0002,car\_

To isolate the next forty characters, use the SUBSTR function:

SUBSTR (ConcatenatedInvoiceInfo, 41, 40)

to get the following string:

parts\_inv0003,car\_parts\_inv0004,car\_par

### Invalid Characters and Replacement Characters Commands

Some formats require a different character set than the one that was used to enter the data in Oracle Applications. For example, some German formats require the output file in ASCII, but the data was entered in German. If there is a mismatch between the original and target character sets you can define an ASCII equivalent to replace the original. For example, you would replace the German umlauted "a" with "ao".

Some formats will not allow certain characters. To ensure that known invalid characters will not be transmitted in your output file, use the invalid characters command to flag occurrences of specific characters.

To use the replacement characters command, specify the source characters in the left column and the replacement characters in the right column. You must enter the source characters in the original character set. This is the only case in a format template in which you use a character set not intended for output. Enter the replacement characters in the required output character set.

For DELIMITER\_BASED formats, if there are delimiters in the data, you can use the escape character "?" to retain their meaning. For example,

First name?+Last name equates to First name+Last name

Which source?? equates to Which source?

Note that the escape character itself must be escaped if it is used in data.

The replacement characters command can be used to support the escape character requirement. Specify the delimiter as the source and the escape character plus the delimiter as the target. For example, the command entry for the preceding examples would be:

---

<REPLACEMENT CHARACTERS>

+	?+
---	----

?	??
---	----

<END REPLACEMENT CHARACTERS>

---

The invalid character command has a single parameter that is a string of invalid characters that will cause the system to error out.

The replacement character process is performed before or during the character set conversion. The character set conversion is performed on the XML extract directly, before the formatting. After the character set conversion, the invalid characters will be checked in terms of the output character set. If no invalid characters are found, the system will proceed to formatting.

### Output Character Set and New Record Character Commands

Use the new record character command to specify the character(s) to delimit the explicit

and implicit record breaks at runtime. Each new record command represents an explicit record break. Each end of table represents an implicit record break. The parameter is a list of constant character names separated by commas.

Some formats contain no record breaks. The generated output is a single line of data. In this case, leave the new record character command parameter field empty.

#### **Number Thousands Separator and Number Decimal Separator**

The default thousands (or group) separator is a comma (",") and the default decimal separator is ".". Use the Number Thousands Separator command and the Number Decimal Separator command to specify separators other than the defaults. For example, to define "." as the group separator and "," as the decimal separator, enter the following:

---

NUMBER THOUSANDS SEPARATOR	.
NUMBER DECIMAL SEPARATOR	,

---

For more information on formatting numbers, see *Format Column*, page 4-13.

## **Expressions, Control Structures, and Functions**

This section describes the rules and usage for expressions in the template. It also describes supported control structures and functions.

### **Expressions**

Expressions can be used in the data column for data fields and some command parameters. An expression is a group of XML extract fields, literals, functions, and operators. Expressions can be nested. An expression can also include the "IF" control structure. When an expression is evaluated it will always generate a result. Side effects are not allowed for the evaluation. Based on the evaluation result, expressions are classified into the following three categories:

- **Boolean Expression** - an expression that returns a boolean value, either true or false. This kind of expression can be used only in the "IF-THEN-ELSE" control structure and the parameter of the display condition command.
- **Numeric Expression** - an expression that returns a number. This kind of expression can be used in numeric data fields. It can also be used in functions and commands that require numeric parameters.
- **Character Expression** - an expression that returns an alphanumeric string. This kind of expression can be used in string data fields (format type Alpha). They can also be used in functions and commands that require string parameters.

## Control Structures

The only supported control structure is "IF-THEN-ELSE". It can be used in an expression. The syntax is:

```
IF <boolean_expressionA> THEN
  <numeric or character expression1>
[ELSIF <boolean_expressionB THEN
  <numeric or character expression2>]
...
[ELSE
  <numeric or character expression3>]
END IF
```

Generally the control structure must evaluate to a number or an alphanumeric string. The control structure is considered to a numeric or character expression. The ELSIF and ELSE clauses are optional, and there can be as many ELSIF clauses as necessary. The control structure can be nested.

The IN predicate is supported in the IF-THEN-ELSE control structure. For example:

```
IF PaymentAmount/Currency/Code IN ('USD', 'EUR', 'AON', 'AZM') THEN

  PayeeAccount/FundsCaptureOrder/OrderAmount/Value * 100
ELSIF PaymentAmount/Currency/Code IN ('BHD', 'IQD', 'KWD') THEN
  PayeeAccount/FundsCaptureOrder/OrderAmount/Value * 1000
ELSE
  PayeeAccount/FundsCaptureOrder/OrderAmount/Value
END IF;
```

## Functions

Following is the list of supported functions:

- **SEQUENCE\_NUMBER** - is a record element index. It is used in conjunction with the Define Sequence command. It has one parameter, which is the sequence defined by the Define Sequence command. At runtime it will increase its sequence value by one each time it is referenced in a record.
- **COUNT** - counts the child level extract instances or child level records of a specific type. Declare the COUNT function on a level above the entity to be counted. The function has one argument. If the argument is a level, the function will count all the instances of the (child) level belonging to the current (parent) level instance.

For example, if the level to be counted is Payment and the current level is Batch, then the COUNT will return the total number of payments in the batch. However, if the current level is RequestHeader, the COUNT will return the total number of payments in the file across all batches. If the argument is a record type, the count function will count all the generated records of the (child level) record type belonging to the current level instance.

- **INTEGER\_PART, DECIMAL\_PART** - returns the integer or decimal portion of a numeric value. This is used in nested expressions and in commands (display condition and group by). For the final formatting of a numeric field in the data

column, use the Integer/Decimal format.

- **IS\_NUMERIC** - boolean test whether the argument is numeric. Used only with the "IF" control structure.
- **TRUNCATE** - truncate the first argument - a string to the length of the second argument. If the first argument is shorter than the length specified by the second argument, the first argument is returned unchanged. This is a user-friendly version for a subset of the SQL substr() functionality.
- **SUM** - sums all the child instance of the XML extract field argument. The field must be a numeric value. The field to be summed must always be at a lower level than the level on which the SUM function was declared.
- **MIN, MAX** - find the minimum or maximum of all the child instances of the XML extract field argument. The field must be a numeric value. The field to be operated on must always be at a lower level than the level on which the function was declared.
- **FORMAT\_DATE** - Formats a date string to any desirable date format. For example:  
`FORMAT_DATE("1900-01-01T18:19:20", "YYYY/MM/DD HH24:MI:SS")`  
will produce the following output:  
1900/01/01 18:19:20
- **FORMAT\_NUMBER** - Formats a number to display in desired format. For example:  
`FORMAT_NUMBER("1234567890.0987654321", "999,999.99")`  
produces the following output:  
1,234,567,890.10
- **MESSAGE\_LENGTH** - returns the length of the message in the EFT message.
- **RECORD\_LENGTH** - returns the length of the record in the EFT message.
- **INSTR** - returns the numeric position of a named character within a text field.
- **SYSDATE, DATE** - gets Current Date and Time.
- **POSITION** - returns the position of a node in the XML document tree structure.
- **REPLACE** - replaces a string with another string.
- **CONVERT\_CASE** - converts a string or a character to UPPER or LOWER case.
- **CHR** - gets the character representation of an argument, which is an ASCII value.

- LPAD, RPAD – generates left or right padding for string values.
- AND, OR, NOT – operator functions on elements.
- Other SQL functions include the following. Use the syntax corresponding to the SQL function.
  - TO\_DATE
  - LOWER
  - UPPER
  - LENGTH
  - GREATEST
  - LEAST
  - DECODE
  - CEIL
  - ABS
  - FLOOR
  - ROUND
  - CHR
  - TO\_CHAR
  - SUBSTR
  - LTRIM
  - RTRIM
  - TRIM
  - IN
  - TRANSLATE

## Identifiers, Operators, and Literals

This section lists the reserved key word and phrases and their usage. The supported



operators are defined and the rules for referencing XML extract fields and using literals.

## Key Words

There are four categories of key words and key word phrases:

- Command and column header key words
- Command parameter and function parameter key words
- Field-level key words
- Expression key words

### Command and Column Header Key Words

The following key words must be used as shown: enclosed in <>s and in all capital letters with a bold font.

- **<LEVEL>**- the first entry of a data table. Associates the table with an XML element and specifies the hierarchy of the table.
- **<END LEVEL>** - declares the end of the current level. Can be used at the end of a table or in a standalone table.
- **<POSITION>** - column header for the first column of data field rows, which specifies the starting position of the data field in a record.
- **<LENGTH>** - column header for the second column of data field rows, which specifies the length of the data field.
- **<FORMAT>** - column header for the third column of data field rows, which specifies the data type and format setting.
- **<PAD>** - column header for the fourth column of data field rows, which specifies the padding style and padding character.
- **<DATA>** - column header for the fifth column of data field rows, which specifies the data source.
- **<COMMENT>** - column header for the sixth column of data field rows, which allows for free form comments.
- **<NEW RECORD>** - specifies a new record.
- **<DISPLAY CONDITION>** - specifies the condition when a record should be printed.
- **<TEMPLATE TYPE>** - specifies the type of the template, either **FIXED\_POSITION\_BASED** or **DELIMITER\_BASED**.

- **<OUTPUT CHARACTER SET>** - specifies the character set to be used when generating the output.
- **<NEW RECORD CHARACTER>** - specifies the character(s) to use to signify the explicit and implicit new records at runtime.
- **<DEFINE LEVEL>** - defines a format-specific level in the template.
- **<BASE LEVEL>** - subcommand for the define level and define concatenation commands.
- **<GROUPING CRITERIA>** - subcommand for the define level command.
- **<END DEFINE LEVEL>** - signifies the end of a level.
- **<DEFINE SEQUENCE>** - defines a record or extract element based sequence for use in the template fields.
- **<RESET AT LEVEL>** - subcommand for the define sequence command.
- **<INCREMENT BASIS>** - subcommand for the define sequence command.
- **<START AT>** - subcommand for the define sequence command.
- **<MAXIMUM>** - subcommand for the define sequence command.
- **<MAXIMUM LENGTH>** - column header for the first column of data field rows, which specifies the maximum length of the data field. For DELIMITER\_BASED templates only.
- **<END DEFINE SEQUENCE>** - signifies the end of the sequence command.
- **<DEFINE CONCATENATION>** - defines a concatenation of child level item that can be referenced as a string the parent level fields.
- **<ELEMENT>** - subcommand for the define concatenation command.
- **<DELIMITER>** - subcommand for the define concatenation command.
- **<END DEFINE CONCATENATION>** - signifies the end of the define concatenation command.
- **<SORT ASCENDING>** - format-specific sorting for the instances of a level.
- **<SORT DESCENDING>** - format-specific sorting for the instances of a level.

#### **Command Parameter and Function Parameter Key Words**

These key words must be entered in all capital letters, nonbold fonts.

- PERIODIC\_SEQUENCE - used in the reset at level subcommand of the define sequence command. It denotes that the sequence number is to be reset outside the template.
- FIXED\_POSITION\_BASED, DELIMITER\_BASED - used in the template type command, specifies the type of template.
- RECORD, LEVEL - used in the increment basis subcommand of the define sequence command. RECORD increments the sequence each time it is used in a new record. LEVEL increments the sequence only for a new instance of the level.

#### **Field-Level Key Words**

- Alpha - in the <FORMAT> column, specifies the data type is alphanumeric.
- Number - in the <FORMAT> column, specifies the data type is numeric.
- Integer - in the <FORMAT> column, used with the Number key word. Takes the integer part of the number. This has the same functionality as the INTEGER function, except the INTEGER function is used in expressions, while the Integer key word is used in the <FORMAT> column only.
- Decimal - in the <FORMAT> column, used with the Number key word. Takes the decimal part of the number. This has the same functionality as the DECIMAL function, except the DECIMAL function is used in expressions, while the Decimal key word is used in the <FORMAT> column only.
- Date - in the <FORMAT> column, specifies the data type is date.
- L, R - in the <PAD> column, specifies the side of the padding (Left or Right).

#### **Expression Key Words**

Key words and phrases used in expressions must be in capital letters and bold fonts.

- IF THEN ELSE IF THEN ELSE END IF - these key words are always used as a group. They specify the "IF" control structure expressions.
- IS NULL, IS NOT NULL - these phrases are used in the IF control structure. They form part of boolean predicates to test if an expression is NULL or not NULL.

#### **Operators**

There are two groups of operators: the boolean test operators and the expression operators. The boolean test operators include: "=", "<>", "<", ">", ">=", and "<=". They can be used only with the IF control structure. The expression operators include: "()", "||", "+", "-", and "\*". They can be used in any expression.

Symbol	Usage
=	Equal to test. Used in the IF control structure only.
<>	Not equal to test. Used in the IF control structure only.
>	Greater than test. Used in the IF control structure only.
<	Less than test. Used in the IF control structure only.
>=	Greater than or equal to test. Used in the IF control structure only.
<=	Less than or equal to test. Used in the IF control structure only.
()	Function argument and expression group delimiter. The expression group inside "()" will always be evaluated first. "()" can be nested.
	String concatenation operator.
+	Addition operator. Implicit type conversion may be performed if any of the operands are not numbers.
-	Subtraction operator. Implicit type conversion may be performed if any of the operands are not numbers.
*	Multiplication operator. Implicit type conversion may be performed if any of the operands are not numbers.
DIV	Division operand. Implicit type conversion may be performed if any of the operands are not numbers. Note that "/" is not used because it is part of the XPATH syntax.

Symbol	Usage
IN	Equal-to-any-member-of test.
NOT IN	Negates the IN operator. Not-Equal-to-any-member-of test.

### Reference to XML Extract Fields and XPATH Syntax

XML elements can be used in any expression. At runtime they will be replaced with the corresponding field values. The field names are case-sensitive.

When the XML extract fields are used in the template, they must follow the XPATH syntax. This is required so that the XML Publisher engine can correctly interpret the XML elements.

There is always an extract element considered as the context element during the XML Publisher formatting process. When XML Publisher processes the data rows in a table, the level element of the table is the context element. For example, when XML Publisher processes the data rows in the Payment table, Payment is the context element. The relative XPATH you use to reference the extract elements are specified in terms of the context element.

For example if you need to refer to the PayeeName element in a Payment data table, you will specify the following relative path:

Payee/PayeeInfo/PayeeName

Each layer of the XML element hierarchy is separated by a backslash "/". You use this notation for any nested elements. The relative path for the immediate child element of the level is just the element name itself. For example, you can use TransactionID element name as is in the Payment table.

To reference a parent level element in a child level table, you can use the "../" notation. For example, in the Payment table if you need to reference the BatchName element, you can specify ../BatchName. The "../" will give you Batch as the context; in that context you can use the BatchName element name directly as BatchName is an immediate child of Batch. This notation goes up to any level for the parent elements. For example if you need to reference the RequesterParty element (in the RequestHeader) in a Payment data table, you can specify the following:

../TrxnParties/RequesterParty

You can always use the absolute path to reference any extract element anywhere in the template. The absolute path starts with a backslash "/". For the PayeeName in the Payment table example above, you will have the following absolute path:

/BatchRequest/Batch/Payment/Payee/PayeeInfo/PayeeName

The absolute path syntax provides better performance.

The identifiers defined by the setup commands such as define level, define sequence

and define concatenation are considered to be global. They can be used anywhere in the template. No absolute or relative path is required. The base level and reset at level for the setup commands can also be specified. XML Publisher will be able to find the correct context for them.

If you use relative path syntax, you should specify it relative to the base levels in the following commands:

- The element subcommand of the define concatenation command
- The grouping criteria subcommand of the define level command

The extract field reference in the start at subcommand of the define sequence command should be specified with an absolute path.

The rule to reference an extract element for the level command is the same as the rule for data fields. For example, if you have a Batch level table and a nested Payment level table, you can specify the Payment element name as-is for the Payment table. Because the context for evaluating the Level command of the Payment table is the Batch.

However, if you skip the Payment level and you have an Invoice level table directly under the Batch table, you will need to specify Payment/Invoice as the level element for the Invoice table.

The XPATH syntax required by the template is very similar to UNIX/LINUX directory syntax. The context element is equivalent to the current directory. You can specify a file relative to the current directory or you can use the absolute path which starts with a "/".

Finally, the extract field reference as the result of the grouping criteria sub-command of the define level command must be specified in single quotes. This tells the XML Publisher engine to use the extract fields as the grouping criteria, not their values.

---

# XSL, SQL, and XSL-FO Support for RTF Templates

## Extended SQL and XSL Functions

XML Publisher has extended a set of SQL and XSL functions for use in RTF templates. The syntax for these extended functions is

`<?xdofx:expression?>`

for extended SQL functions or

`<?xdoxslt:expression?>`

for extended XSL functions.

**Note:** You cannot mix xdofx statements with XSL expressions in the same context. For example, assume you had two elements, FIRST\_NAME and LAST\_NAME that you wanted to concatenate into a 30-character field and right pad the field with the character "x", you could NOT use the following:

**INCORRECT:**

`<?xdofx:rpadd(concat(FIRST_NAME, LAST_NAME), 30, 'x')?>`

because `concat` is an XSL expression. Instead, you could use the following:

**CORRECT:**

`<?xdofx:rpadd(FIRST_NAME || LAST_NAME, 30, 'x')?>`

The supported functions are shown in the following table:

SQL Statement or XSL Expression	Usage	Description
2+3	<?xdofx:2+3?>	Addition
2-3	<?xdofx:2-3?>	Subtraction
2*3	<?xdofx:2*3?>	Multiplication
2/3	<?xdofx:2/3?>	Division
2**3	<?xdofx:2**3?>	Exponential
3  2	<?xdofx:3  2?>	Concatenation
lpad('aaa',10,'.')	<?xdofx:lpad('aaa',10,'.')?>	<p>The lpad function pads the left side of a string with a specific set of characters. The syntax for the lpad function is:</p> <pre>lpad (   string1,padded_length,[pad_string] )</pre> <p><i>string1</i> is the string to pad characters to (the left-hand side).</p> <p><i>padded_length</i> is the number of characters to return.</p> <p><i>pad_string</i> is the string that will be padded to the left-hand side of <i>string1</i>.</p>
rpadd('aaa',10,'.')	<?xdofx:rpadd('aaa',10,'.')?>	<p>The rpadd function pads the right side of a string with a specific set of characters.</p> <p>The syntax for the rpadd function is:</p> <pre>rpadd (   string1,padded_length,[pad_string] ).</pre> <p><i>string1</i> is the string to pad characters to (the right-hand side).</p> <p><i>padded_length</i> is the number of characters to return.</p> <p><i>pad_string</i> is the string that will be padded to the right-hand side of <i>string1</i>.</p>



SQL Statement or XSL Expression	Usage	Description
<code>decode('xxx','bbb','ccc','xxx','ddd')</code>	<code>&lt;?xdoxf:decode('xxx','bbb','ccc','xxx','ddd')?&gt;</code>	<p>The <code>decode</code> function has the functionality of an IF-THEN-ELSE statement. The syntax for the <code>decode</code> function is:</p> <pre>decode(expression, search, result [,search, result]...[, default])</pre> <p><i>expression</i> is the value to compare.</p> <p><i>search</i> is the value that is compared against <i>expression</i>.</p> <p><i>result</i> is the value returned, if <i>expression</i> is equal to <i>search</i>.</p> <p><i>default</i> is returned if no matches are found.</p>
<code>Instr('abcabcabc','a',2)</code>	<code>&lt;?xdoxf:Instr('abcabcabc','a',2)?&gt;</code>	<p>The <i>instr</i> function returns the location of a substring in a string. The syntax for the <i>instr</i> function is:</p> <pre>instr(string1,string2,[start_position],[nth_appearance])</pre> <p><i>string1</i> is the string to search.</p> <p><i>string2</i> is the substring to search for in <i>string1</i>.</p> <p><i>start_position</i> is the position in <i>string1</i> where the search will start. The first position in the string is 1. If the <i>start_position</i> is negative, the function counts back <i>start_position</i> number of characters from the end of <i>string1</i> and then searches towards the beginning of <i>string1</i>.</p> <p><i>nth appearance</i> is the <i>nth</i> appearance of <i>string2</i>.</p>
<code>substr('abcdefg',2,3)</code>	<code>&lt;?xdoxf:substr('abcdefg',2,3)?&gt;</code>	<p>The <i>substr</i> function allows you to extract a substring from a string. The syntax for the <i>substr</i> function is:</p> <pre>substr(string, start_position, [length])</pre> <p><i>string</i> is the source string.</p> <p><i>start_position</i> is the position for extraction. The first position in the string is always 1.</p> <p><i>length</i> is the number of characters to extract.</p>

SQL Statement or XSL Expression	Usage	Description
replace(name,'John','Jon')	<?xdoxf:replace(name,'John','Jon')?>	<p>The replace function replaces a sequence of characters in a string with another set of characters. The syntax for the replace function is:</p> <pre>replace(string1,string_to_replace,[replacement_string])</pre> <p><i>string1</i> is the string to replace a sequence of characters with another set of characters.</p> <p><i>string_to_replace</i> is the string that will be searched for in <i>string1</i>.</p> <p><i>replacement_string</i> is optional. All occurrences of <i>string_to_replace</i> will be replaced with <i>replacement_string</i> in <i>string1</i>.</p>
to_number('12345')	<?xdoxf:to_number('12345')?>	Function to_number converts char, a value of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype containing a number in the format specified by the optional format model <i>fmt</i> , to a value of NUMBER datatype.
to_char(12345)	<?xdoxf:to_char('12345')?>	Use the TO_CHAR function to translate a value of NUMBER datatype to VARCHAR2 datatype.
to_date	<?xdoxf:to_date ( char [, <i>fmt</i> [, 'nlsparam']] )	TO_DATE converts char of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to a value of DATE datatype. The <i>fmt</i> is a date format specifying the format of <i>char</i> . If you omit <i>fmt</i> , then <i>char</i> must be in the default date format. If <i>fmt</i> is 'J', for Julian, then <i>char</i> must be an integer.
sysdate()	<?xdoxf:sysdate()?>	SYSDATE returns the current date and time. The datatype of the returned value is DATE. The function requires no arguments.
minimum	<?xdoxslt:minimum(ELEMENT_NAME)?>	Returns the minimum value of the element in the set.
maximum	<?xdoxslt:maximum(ELEMENT_NAME)?>	Returns the maximum value of the element in the set.

SQL Statement or XSL Expression	Usage	Description
chr	<?xdoxf:chr( <i>n</i> )?>	CHR returns the character having the binary equivalent to <i>n</i> in either the database character set or the national character set.
ceil	<?xdoxf:ceil( <i>n</i> )?>	CEIL returns smallest integer greater than or equal to <i>n</i> .
floor	<?xdoxf:floor( <i>n</i> )?>	FLOOR returns largest integer equal to or less than <i>n</i> .
round	<?xdoxf:round ( <i>number</i> [, <i>integer</i> ] )?>	ROUND returns <i>number</i> rounded to <i>integer</i> places right of the decimal point. If <i>integer</i> is omitted, then <i>number</i> is rounded to 0 places. <i>integer</i> can be negative to round off digits left of the decimal point. <i>integer</i> must be an integer.
lower	<?xdoxf:lower ( <i>char</i> )?>	LOWER returns <i>char</i> , with all letters lowercase. <i>char</i> can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The return value is the same datatype as <i>char</i> .
upper	<?xdoxf:upper( <i>char</i> )?>	UPPER returns <i>char</i> , with all letters uppercase. <i>char</i> can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The return value is the same datatype as <i>char</i> .
length	<?xdoxf:length( <i>char</i> )?>	The "length" function returns the length of <i>char</i> . LENGTH calculates length using characters as defined by the input character set.
greatest	<?xdoxf:greatest ( <i>expr</i> [, <i>expr</i> ]... )?>	GREATEST returns the greatest of the list of <i>exprs</i> . All <i>exprs</i> after the first are implicitly converted to the datatype of the first <i>expr</i> before the comparison.
least	<?xdoxf:least ( <i>expr</i> [, <i>expr</i> ]... )?>	LEAST returns the least of the list of <i>exprs</i> . All <i>exprs</i> after the first are implicitly converted to the datatype of the first <i>expr</i> before the comparison.

The following table shows supported combination functions:

SQL Statement	Usage
$(2+3/4-6*7)/8$	<code>&lt;?xdofx:(2+3/4-6*7)/8?&gt;</code>
<code>lpad(substr('1234567890',5,3),10,'^')</code>	<code>&lt;?xdofx:lpad(substr('1234567890',5,3),10,'^')?&gt;</code>
<code>decode('a','b','c','d','e','1')  instr('321',1,1)</code>	<code>&lt;?xdofx:decode('a','b','c','d','e','1')  instr('321',1,1)?&gt;</code>

## XSL Equivalents

The following table lists the XML Publisher simplified syntax with the XSL equivalents.

Supported XSL Elements	Description	XML Publisher Syntax
<code>&lt;xsl:value-of select="name"&gt;</code>	Placeholder syntax	<code>&lt;?name?&gt;</code>
<code>&lt;xsl:apply-templates select="name"&gt;</code>	Applies a template rule to the current element's child nodes.	<code>&lt;?apply:name?&gt;</code>
<code>&lt;xsl:copy-of select="name"&gt;</code>	Creates a copy of the current node.	<code>&lt;?copy-of:name?&gt;</code>
<code>&lt;xsl:call-template name="name"&gt;</code>	Calls a named template to be inserted into/applied to the current template.	<code>&lt;?call:name?&gt;</code>
<code>&lt;xsl:sort select="name"&gt;</code>	Sorts a group of data based on an element in the dataset.	<code>&lt;?sort:name?&gt;</code>
<code>&lt;xsl:for-each select="name"&gt; &gt;</code>	Loops through the rows of data of a group, used to generate tabular output.	<code>&lt;?for-each:name?&gt;</code>
<code>&lt;xsl:choose&gt;</code>	Used in conjunction with <code>when</code> and <code>otherwise</code> to express multiple conditional tests.	<code>&lt;?choose?&gt;</code>

Supported XSL Elements	Description	XML Publisher Syntax
<code>&lt;xsl:when test="exp"&gt;</code>	Used in conjunction with <code>choose</code> and <code>otherwise</code> to express multiple conditional tests	<code>&lt;?when:expression?&gt;</code>
<code>&lt;xsl:otherwise&gt;</code>	Used in conjunction with <code>choose</code> and <code>when</code> to express multiple conditional tests	<code>&lt;?otherwise?&gt;</code>
<code>&lt;xsl:if test="exp"&gt;</code>	Used for conditional formatting.	<code>&lt;?if:expression?&gt;</code>
<code>&lt;xsl:template name="name"&gt;</code>	Template declaration	<code>&lt;?template:name?&gt;</code>
<code>&lt;xsl:variable name="name"&gt;</code>	Local or global variable declaration	<code>&lt;?variable:name?&gt;</code>
<code>&lt;xsl:import href="url"&gt;</code>	Import the contents of one stylesheet into another	<code>&lt;?import:url?&gt;</code>
<code>&lt;xsl:include href="url"&gt;</code>	Include one stylesheet in another	<code>&lt;?include:url?&gt;</code>
<code>&lt;xsl:stylesheet xmlns:x="url"&gt;</code>	Define the root element of a stylesheet	<code>&lt;?namespace:x=url?&gt;</code>

## Using FO Elements

You can use most FO elements in an RTF template inside the Microsoft Word form fields. The following FO elements have been extended for use with XML Publisher RTF templates. The XML Publisher syntax can be used with either RTF template method.

The full list of FO elements supported by XML Publisher can be found in the Appendix: Supported XSL-FO Elements, page A-1.

FO Element	XML Publisher Syntax
<code>&lt;fo:page-number-citation ref-id="id"&gt;</code>	<code>&lt;?fo:page-number-citation:id?&gt;</code>
<code>&lt;fo:page-number&gt;</code>	<code>&lt;?fo:page-number?&gt;</code>
<code>&lt;fo:ANY NAME WITHOUT ATTRIBUTE&gt;</code>	<code>&lt;?fo:ANY NAME WITHOUT ATTRIBUTE?&gt;</code>



---

## Adding Template Translations

### Translatable Templates

XML Publisher supports the addition of translation files for RTF templates. Depending on your implementation (JD Edwards EnterpriseOne or PeopleSoft Enterprise) the method for defining a template as translatable may differ. Please see your product documentation.

When you define a template as translatable, XML Publisher extracts the translatable strings and exports them to an XLIFF (.xlf) file. This XLIFF file can then be sent to a translation provider, or using a text editor, you can enter the translation for each string.

**Note:** XLIFF is the XML Localization Interchange File Format. It is the standard format used by localization providers. For more information about the XLIFF specification, see <http://www.oasis-open.org/committees/xliff/documents/xliff-specification.htm>

A "translatable string" is any text in the template that is intended for display in the published report, such as table headers and field labels. Text supplied at runtime from the data is not translatable, nor is any text that you supply in the Microsoft Word form fields.

You can translate the template XLIFF file into as many languages as desired and then associate these translations to the original template.

### Working with XLIFF Files

This section describes the structure of the XLIFF file and how to update the file with the desired translation.

#### Structure of the XLIFF File

The XLIFF file generated by BI Publisher has the following structure:

```

<xliff>
  <file>
    <header>
    <body>
      <trans-unit>
        <source>
        <target>
        <note>

```

The following figure shows an excerpt from an untranslated XLIFF file:

```

<?xml version = '1.0' encoding = 'utf-8' ?>
<xliff version="1.0">
  <file source-language="en-US" target-language="en-US" datatype="XDO" original="orpher">
    <header/>
    <body>
      <trans-unit id="d678c24b" maxbytes="4000" maxwidth="90" size-unit="char" translatable="yes">
        <source>Italian Purchase VAT Register - [&1]</source>
        <target>Italian Purchase VAT Register - [&1]</target>
        <note>Text located: header/table, token &1: anonymous placeholder(s)</note>
      </trans-unit>
      <trans-unit id="4d3eb24" maxbytes="4000" maxwidth="15" size-unit="char" translatable="yes">
        <source>Total</source>
        <target>Total</target>
        <note>Text located: body/table</note>
      </trans-unit>
      <trans-unit id="aec17e" maxbytes="4000" maxwidth="37" size-unit="char" translatable="yes">
        <source>Non-Recoverable</source>
        <target>Non-Recoverable</target>
        <note>Text located: body/table</note>
      </trans-unit>
      .
      .
    </body>
  </file>
</xliff>

```

## source-language and target-language attributes

The `<file>` element includes the attributes `source-language` and `target-language`. The valid value for `source-language` and `target-language` is a combination of the language code and country code as follows:

- the two-letter ISO 639 language code
- the two-letter ISO 3166 country code

**Note:** For more information on the International Organization for Standardization (ISO) and the code lists, see International Organization for Standardization [<http://www.iso.org/iso/en/ISOOnline.frontpage>].

For example, the value for English-United States is "en-US". This combination is also referred to as a *locale*.

When you edit the exported XLIFF file you must change the `target-language` attribute to the appropriate locale value of your target language. The following table



shows examples of source-language and target-language attribute values appropriate for the given translations:

Translation (Language/Territory)	source-language value	target-language value
From English/US To English/Canada	en-US	en-CA
From English/US To Chinese/China	en-US	zh-CN
From Japanese/Japan To French/France	ja-JP	fr-FR

### Embedded Data Fields

Some templates contain placeholders for data fields embedded in the text display strings of the report. For example, the title of the sample report is

#### Italian Purchase VAT Register - (year)

where (year) is a placeholder in the RTF template that will be populated at runtime by data from an XML element. These fields are not translatable, because the value comes from the data at runtime.

To identify embedded data fields, the following token is used in the XLIFF file:

[ & n ]

where *n* represents the numbered occurrence of a data field in the template.

For example, in the preceding XLIFF sample, the first translatable string is

```
<source>Italian Purchase VAT Register - [ & 1 ]</source>
```

**Warning:** Do not edit or delete the embedded data field tokens or you will affect the merging of the XML data with the template.

### <source> and <target> Elements

Each <source> element contains a translatable string from the template in the source language of the template. For example,

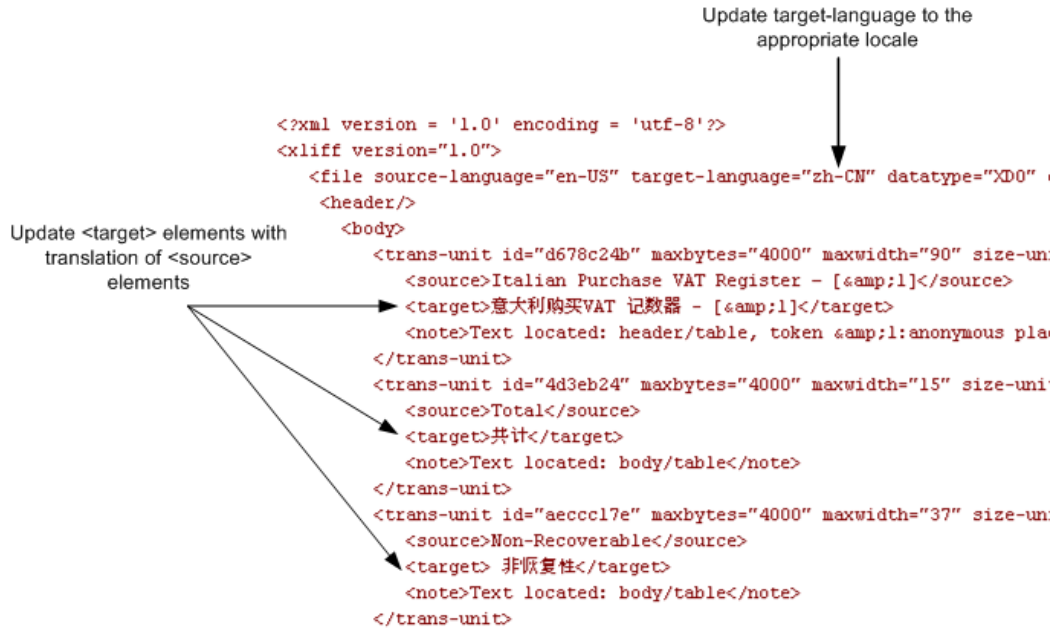
```
<source>Total</source>
```

When you initially export the XLIFF file for translation, the source and target elements are all identical. To create the translation for this template, enter the appropriate translation for each source element string in its corresponding <target> element.

Therefore if you were translating the sample template into German, you would enter the following for the Total string:

```
<source>Total</source>
<target>Gesamtbetrag</target>
```

The following figure shows the sample XLIFF file from the previous figure updated with the Chinese translation:



#### Uploading the Translated XLIFF Files

Ensure that the file is saved in UTF-8 (the xml encoding attribute must equal 'utf-8').

See your product-specific documentation for procedures on uploading and associating the translated XLIFF files with the base template.

---

# Setting Runtime Properties

## Setting Properties in a Configuration File

You can customize the behavior of XML Publisher by setting properties in a configuration file. The configuration file is optional. No default configuration file is provided.

The file is primarily used for:

- Setting a temporary directory
- Setting properties for PDF, RTF, and HTML output files
- Setting PDF security properties
- Setting font locations and substitutions
- Setting translation properties

**Important:** It is **strongly** recommended that you set up this configuration file to create a temporary directory for processing large files. If you do not, you will encounter "Out of Memory" errors when processing large files. Create a temporary directory by setting the `system-temp-dir` property.

It is also recommended that you secure the configuration file if you use it to set the PDF security passwords.

## File Name and Location

The configuration file is named `xdo.cfg`.

The file is located under the `<JRE_TOP>/jre/lib`, for example: `jdk/jre/lib`.

## Namespace

The namespace for this configuration file is:

`http://xmlns.oracle.com/oxp/config/`

## Configuration File Example

Following is a sample configuration file:

```
<config version="1.0.0"
  xmlns="http://xmlns.oracle.com/oxp/config/"><!-- Properties -->
<properties>
  <!-- System level properties -->
  <property name="system-temp-dir">/tmp</property>

  <!-- PDF compression -->
  <property name="pdf-compression">true</property>

  <!-- PDF Security -->
  <property name="pdf-security">true</property>
  <property name="pdf-open-password">user</property>
  <property name="pdf-permissions-password">owner</property>
  <property name="pdf-no-printing">true</property>
  <property name="pdf-no-changing-the-document">true</property>
</properties>

<!-- Font setting -->
<font>
  <!-- Font setting (for FO to PDF etc...) -->
  <font family="Arial" style="normal" weight="normal">
    <truetype path="/fonts/Arial.ttf" />
  </font>
  <font family="Default" style="normal" weight="normal">
    <truetype path="/fonts/ALBANWTJ.ttf" />
  </font>

  <!--Font substitute setting (for PDFForm filling etc...) -->
  <font-substitute name="MSGothic">
    <truetype path="/fonts/msgothic.ttc" ttcno="0" />
  </font-substitute>
</font>
</config>
```

## How to Read the Element Specifications

The following is an example of an element specification:

```
<Element Name Attribute1="value"
  Attribute2="value"
  AttributeN="value"
  <Subelement Name1/>[occurrence-spec]
  <Subelement Name2>...</Subelement Name2>
  <Subelement NameN>...</Subelement NameN>
</Element Name>
```

The `[occurrence-spec]` describes the cardinality of the element, and corresponds to the following set of patterns:

- [0..1] - indicates the element is optional, and may occur only once.
- [0..n] - indicates the element is optional, and may occur multiple times.

## Structure

The `<config>` element is the root element. It has the following structure:

```
<config version="cdata" xmlns="http://xmlns.oracle.com/oxp/config/">
  <font> ... </font> [0..n]
  <properties> ... </properties> [0..n]
</config>
```

## Attributes

<b>version</b>	The version number of the configuration file format. Specify 1.0.0.
<b>xmlns</b>	The namespace for XML Publisher's configuration file. Must be <code>http://xmlns.oracle.com/oxp/config/</code>

## Description

The root element of the configuration file. The configuration file consists of two parts:

- Properties (`<properties>` elements)
- Font definitions (`<font>` elements)

The `<font>` and `<properties>` elements can appear multiple times. If conflicting definitions are set up, the last occurrence prevails.

## Properties

This section describes the `<properties>` element and the `<property>` element.

### The `<properties>` element

The properties element is structured as follows:

```
<properties locales="cdata">
  <property>...
</property> [0..n]
</properties>
```

## Description

The `<properties>` element defines a set of properties. You can specify the `locales` attribute to define locale-specific properties. Following is an example:

### Example

```
<!-- Properties for all locales -->
<properties>...Property definitions here...
</properties>

<!--Korean specific properties-->
<properties locales="ko-KR">
  ...Korean-specific property definitions here...
</properties>
```

## The <property> element

The <property> element has the following structure:

```
<property name="cdata"> ...pcdata...
</property>
```

### Attributes

<b>name</b>	Specify the property name.
-------------	----------------------------

### Description

Property is a name-value pair. Specify the internal property name (key) to the name attribute and the value to the element value. See List of Properties, page 7-4 for the list of the internal property names.

### Example

```
<properties>
  <property name="system-temp-dir">d:\tmp</property>
  <property name="system-cache-page-size">50</property>
  <property name="pdf-replace-smart-quotes">>false</property>
</properties>
```

## List of Properties

The following tables list the available properties. They are organized into the following groups:

- General Properties
- PDF Output Properties
- PDF Security
- RTF Output
- HTML Output
- FO Processing Properties
- PDF Template Properties

- RTF Template Properties
- PDF Document Merger Property
- XLIFF Extraction Properties

## General Properties

General properties are shown in the following table:

Property Name	Default Value	Description
system-temp-dir	N/A	Enter the directory path for the temporary directory to be used by the FO Processor when processing large files. It is strongly recommended that you set a temporary directory to avoid "Out of Memory" errors.

## PDF Output Properties

The following table shows properties supported for PDF output:

Property Name	Default Value	Description
pdf-compression	True	Specify "True" or "False" to control compression of the output PDF file.
pdf-hide-menubar	False	Specify "True" to hide the viewer application's menu bar when the document is active.
pdf-hide-toolbar	False	Specify "True" to hide the viewer application's toolbar when the document is active.
pdf-replace-smart quotes	True	Set to "False" if you do not want curly quotes replaced with straight quotes in your PDF output.

## PDF Security

Use the following properties to control the security settings for your output PDF documents:

Property Name	Default Value	Description
pdf-security	False	<p>If you specify "True," the output PDF file will be encrypted. You must also specify the following properties:</p> <ul style="list-style-type: none"> <li>pdf-open-password</li> <li>pdf-permissions-password</li> <li>pdf-encryption-level</li> </ul>
pdf-open-password	N/A	<p>This password will be required for opening the document. It will enable users to open the document only. This property is enabled only when pdf-security is set to "True".</p>
pdf-permissions-password	N/A	<p>This password enables users to override the security setting. This property is effective only when pdf-security is set to "True".</p>



Property Name	Default Value	Description
pdf-encryption-level	0 - low	<p>Specify the encryption level for the output PDF file. The possible values are:</p> <ul style="list-style-type: none"> <li>0: Low (40-bit RC4, Acrobat 3.0 or later)</li> <li>1: High (128-bit RC4, Acrobat 5.0 or later)</li> </ul> <p>This property is effective only when pdf-security is set to "True". When pdf-encryption-level is set to 0, you can also set the following properties:</p> <ul style="list-style-type: none"> <li>pdf-no-printing</li> <li>pdf-no-changing-the-document</li> <li>pdf-no-cceda</li> <li>pdf-no-accff</li> </ul> <p>When pdf-encryption-level is set to 1, the following properties are available:</p> <ul style="list-style-type: none"> <li>pdf-enable-accessibility</li> <li>pdf-enable-copying</li> <li>pdf-changes-allowed</li> <li>pdf-printing-allowed</li> </ul>
pdf-no-printing	False	Permission available when pdf-encryption-level is set to 0. When set to "True", printing is disabled for the PDF file.
pdf-no-changing-the-document	False	Permission available when pdf-encryption-level is set to 0. When set to "True", the PDF file cannot be edited.
pdf-no-cceda	False	Permission available when pdf-encryption-level is set to 0. When set to "True", the context copying, extraction, and accessibility features are disabled.
pdf-no-accff	False	Permission available when pdf-encryption-level is set to 0. When set to "True", the ability to add or change comments and form fields is disabled.

Property Name	Default Value	Description
pdf-enable-accessibility	True	Permission available when pdf-encryption-level is set to 1. When set to "True", text access for screen reader devices is enabled.
pdf-enable-copying	False	Permission available when pdf-encryption-level is set to 1. When set to "True", copying of text, images, and other content is enabled.
pdf-changes-allowed	0	<p>Permission available when pdf-encryption-level is set to 1. Valid Values are:</p> <ul style="list-style-type: none"> <li>• 0: none</li> <li>• 1: Allows inserting, deleting, and rotating pages</li> <li>• 2: Allows filling in form fields and signing</li> <li>• 3: Allows commenting, filling in form fields, and signing</li> <li>• 4: Allows all changes except extracting pages</li> </ul>
pdf-printing-allowed	0	<p>Permission available when pdf-encryption-level is set to 1. Valid values are:</p> <ul style="list-style-type: none"> <li>• 0: None</li> <li>• 1: Low resolution (150 dpi)</li> <li>• 2: High resolution</li> </ul>

## RTF Output

The following properties can be set to govern RTF output files:

Property Name	Default Value	Description
rtf-track-changes	False	Set to "True" to enable change tracking in the output RTF document.

Property Name	Default Value	Description
<code>rtf-protect-document-for-tracked-changes</code>	False	Set to "True" to protect the document for tracked changes.
<code>rtf-output-default-font</code>	Times New Roman:8	When generating an RTF output file, there may be empty table cells or empty lines where the RTF generator cannot set the font name and size information. When the document is opened in MS Word, these areas are assigned a default font and font size. To change the default setting for these occurrences, set this property as follows: "font name:font size", for example: "Arial:12".

## HTML Output

The following properties can be set to govern HTML output files:

Property Name	Default Value	Description
<code>html-image-base-uri</code>	N/A	Base URI which is inserted into the <code>src</code> attribute of the image tag before the image file name. This works only when the image is embedded in the template.
<code>html-image-dir</code>	N/A	Enter the directory for XML Publisher to store the image files that are embedded in the template.
<code>html-css-base-uri</code>	N/A	Base URI which is inserted into the HTML header to specify where the cascading stylesheets (CSS) for your output HTML documents will reside. You must set this property when <code>make-accessible</code> is true.
<code>html-css-dir</code>	N/A	The CSS directory where XML Publisher stores the css file. You must set this property when <code>make-accessible</code> is true.
<code>html-show-header</code>	True	Set to "False" to suppress the template header in HTML output.
<code>html-show-footer</code>	True	Set to "False" to suppress the template footer in HTML output.

Property Name	Default Value	Description
html-replace-smartquotes	True	Set to "False" if you do not want curly quotes replaced with straight quotes in your HTML output.
html-output-charset	UTF-8	Specify the output HTML character set.
make-accessible	False	Specify true if you want to make the HTML output accessible.

## FO Processing Properties

The following properties can be set to govern FO processing:

Property Name	Default Value	Description
digit-substitution	None	Valid values are "None" and "National". When set to "None", Eastern European numbers will be used. When set to "National", Hindi format (Arabic-Indic digits) will be used. This setting is effective only when the locale is Arabic, otherwise it is ignored.
system-cache-page-size	50	This property is enabled only when you have specified a Temporary Directory (under General properties). During table of contents generation, the FO Processor caches the pages until the number of pages exceeds the value specified for this property. It then writes the pages to a file in the Temporary Directory.
fo-prevent-variable-header	False	If "True", prevents variable header support. Variable header support automatically extends the size of the header to accommodate the contents.
fo-merge-conflict-resolution	False	When merging multiple XSL-FO inputs, the FO Processor automatically adds random prefixes to resolve conflicting IDs. Setting this property to "True" disables this feature.
xslt-xdoparser	True	Controls XML Publisher's parser usage. If set to False, XSLT will not be parsed.

Property Name	Default Value	Description
<code>xslt-scalable</code>	False	Controls the scalable feature of the XDO parser. The property <code>xslt-xdoparser</code> must be set to "True" for this property to be effective.
<code>rtf-adj-table-border-overlap</code>	False	Sets the behavior of the top/bottom borders of adjacent tables. If you want the top and bottom borders of adjacent tables to overlap (for example, if you want to repeat a table multiple times and make the whole repeating area appear to be a single table), set this property to "true."

## RTF Template Properties

The following properties can be set to govern RTF templates:

Property Name	Default Value	Description
<code>rtf-extract-attribute-sets</code>	Auto	<p>The RTF processor will automatically extract attribute sets within the generated XSL-FO. The extracted sets are placed in an extra FO block, which can be referenced. This improves processing performance and reduces file size.</p> <p>Valid values are:</p> <ul style="list-style-type: none"> <li>• Enable - extract attribute sets for all templates and subtemplates</li> <li>• Auto - extract attribute sets for templates, but not subtemplates</li> <li>• Disable - do not extract attribute sets</li> </ul>
<code>rtf-rewrite-path</code>	True	When converting an RTF template to XSL-FO, the RTF processor will automatically rewrite the XML tag names to represent the full XPath notations. Set this property to "False" to disable this feature.

Property Name	Default Value	Description
rtf-checkbox-glyph	Default value: Albany WT J;9746;9747/A	<p>The XML Publisher default PDF output font does not include a glyph to represent a checkbox. If your template contains a checkbox, use this property to define a Unicode font for the representation of checkboxes in your PDF output. You must define the Unicode font number for the "checked" state and the Unicode font number for the "unchecked" state using the following syntax: <code>fontname;&lt;unicode font number for true value's glyph&gt;;&lt;unicode font number for false value's glyph&gt;</code></p> <p>Example: Albany WT J;9746;9747/A</p> <p>Note that the font that you specify must be made available to XML Publisher at runtime.</p>

## PDF Template Properties

The following properties can be set to configure form field behavior in PDF output files generated from PDF templates:

Property Name	Default Value	Description
remove-pdf-fields	false	Specify "true" to remove PDF fields from the output. When PDF fields are removed, data entered in the fields cannot be extracted. For more information, see <i>Setting Fields as Updateable or Read Only</i> , page 3-15.
all-field-readonly	true	By default, XML Publisher sets all fields in the output PDF of a PDF template to be read only. If you want to set all fields to be updateable, set this property to "false". For more information, see <i>Setting Fields as Updateable or Read Only</i> , page 3-15.

Property Name	Default Value	Description
<code>all-fields-readonly-asis</code>	false	Set this property to "true" if you want to maintain the "Read Only" setting of each field as defined in the PDF template. This property overrides the settings of <code>all-field-readonly</code> . For more information, see Setting Fields as Updateable or Read Only, page 3-15.

## PDF Document Merger Property

Property Name	Default Value	Description
<code>pdf-tempfile-max-size</code>	unlimited	This property sets the maximum size for the temporary file used during batch processing by the PDF Document Merger. Enter the maximum size in bytes. For more information, see PDF Document Merger, page 8-30.

## XLIFF Extraction

The following properties can be set to govern XLIFF extraction:

Property Name	Default Value	Description
<code>xliff-trans-expansion</code>	150 (percentage)	This property determines the maximum percent expansion of an extracted translation unit. For example, if set to 200, the XLIFF extractor will allow expansion by 200% - that is, a 10-character element will have a maximum width of 30 characters.
<code>xliff-trans-min-length</code>	15 (characters)	Sets a minimum length in characters for the extracted translation unit. For example, the default expansion of a 4-character field is 10 characters (based on the default setting of Translation expansion percentage of 150). If the Minimum translation length is 15, this field will be reset to 15 characters.

Property Name	Default Value	Description
<code>xliff-trans-max-length</code>	4000 (characters)	Sets a limit to the calculated expansion of the translation unit (in characters). For example, the default maximum expansion of 100 characters is 250 characters. Setting Maximum translation length to 200 would limit this expansion to 200 characters.
<code>xliff-trans-null</code>	False	Instructs the XLIFF extractor to create a translation unit for a record that contains only spaces (is null). Set to "True" to generate the translation unit.
<code>xliff-trans-symbol</code>	False	Instructs the XLIFF extractor whether to extract symbol characters. If set to "False" only A-Z and a-z will be extracted.
<code>xliff-trans-keyword</code>	True	If set to "False", words with underscores will not be extracted.

## Font Definitions

Font definitions include the following elements:

- `<font>`
- `<font-substitute>`
- `<truetype>`
- `<type1>`

For the list of Truetype and Type1 fonts, see *Predefined Fonts*, page 7-18.

### `<font>` element

The `<font>` element is structured as follows:

```
<font locales="cdata">
  <font> ... </font> [0..n]
  <font-substitute> ... </font-substitute> [0..n]
</font>
```

#### Attributes

**locales** Specify the locales for this font definition. This attribute is



optional.

### Description

The `<font>` element defines a set of fonts. Specify the `locales` attribute to define locale-specific fonts.

#### Example

```
<!-- Font definitions for all locales -->
<font>
  ..Font definitions here...
</font>

<!-- Korean-specific font definitions -->
<font locales="ko-KR">
  ... Korean Font definitions here...
</font>
```

### `<font>` element

Following is the structure of the `<font>` element:

```
<font family="cdata" style="normalitalic"
weight="normalbold">
  <truetype>...</truetype>
or <typel> ... <typel>
</font>
```

### Attributes

<b>family</b>	Specify any family name for the font. If you specify "Default" for this attribute, you can define a default fallback font. The <b>family</b> attribute is case-insensitive.
<b>style</b>	Specify "normal" or "italic" for the font style.
<b>weight</b>	Specify "normal" or "bold" for the font weight.

### Description

Defines a XML Publisher font. This element is primarily used to define fonts for FO-to-PDF processing (RTF to PDF). The PDF Form Processor (used for PDF templates) does not refer to this element.

#### Example

```
<!-- Define "Arial" font -->
<font family="Arial" style="normal" weight="normal">
  <truetype path="/fonts/Arial.ttf"/>
</font>
```

### `<font-substitute>` element

Following is the structure of the `font-substitute` element:

```
<font-substitute name="cdata">
  <truetype>...</truetype>
or <type1>...</type1>
</font-substitute>
```

#### Attributes

**name** Specify the name of the font to be substituted.

#### Description

Defines a font substitution. This element is used to define fonts for the PDF Form Processor.

##### Example

```
<font-substitute name="MSGothic">
  <truetype path="/fonts/msgothic.ttc" ttccno=0"/>
</font-substitute>
```

### <type1> element

The form of the <type1> element is as follows:

```
<type1 name="cdata"/>
```

#### Attributes

**name** Specify one of the Adobe standard Latin1 fonts, such as "Courier".

#### Description

<type1> element defines an Adobe Type1 font.

##### Example

```
<!--Define "Helvetica" font as "Serif" -->
<font family="serif" style="normal" weight="normal">
  <type1 name="Helvetica"/>
</font>
```

## Locales

A locale is a combination of an ISO language and an ISO country. ISO languages are defined in ISO 639 and ISO countries are defined in ISO 3166.

The structure of the locale statement is

ISO Language-ISO country

Locales are not case-sensitive and the ISO country can be omitted.

Example locales:

- en

- en-US
- EN-US
- ja
- ko
- zh-CN

## Font Fallback Logic

XML Publisher uses a font mapping fallback logic so that the result font mappings used for a template are a composite of the font mappings from the template up to the site level. If a mapping is found for a font on more than one level, the most specific level's value overrides the others.

The resulting font mapping to use in any particular instance is the sum of all the applicable font mappings. The applicable mappings in order of preference, are:

Language + Territory match, territory null > Language + Territory null (global value)

For example:

Suppose for a particular template, there are different font mapping sets assigned at the site and template levels, with the mappings shown in the following table:

Level	Font Family	Style	Weight	Language	Territory	Target Font
Site	Times New Roman	normal	normal	(none)	(none)	Times
Site	Arial	normal	normal	Japanese	Japan	Times
Template	Arial	normal	normal	Japanese	(none)	Courier
Template	Trebuchet MS	normal	normal	(none)	(none)	Helvetica

At runtime if the locale of the template file is Japanese/Japan, the following font mappings will be used:

Font Family	Style	Weight	Target Font
Times New Roman	normal	normal	Times
Arial	normal	normal	Times
Trebuchet MS	normal	normal	Helvetica

Note that even though there is a mapping for Arial at the template level, the site level value is used because it has a better match for the locale.

## Predefined Fonts

XML Publisher provides a set of Type1 fonts and a set of TrueType fonts. You can select any of these fonts as a target font with no additional setup required.

The Type1 fonts are listed in the following table:

### *Type 1 Fonts*

Number	Font Family	Style	Weight	Font Name
1	serif	normal	normal	Time-Roman
1	serif	normal	bold	Times-Bold
1	serif	italic	normal	Times-Italic
1	serif	italic	bold	Times-BoldItalic
2	sans-serif	normal	normal	Helvetica
2	sans-serif	normal	bold	Helvetica-Bold
2	sans-serif	italic	normal	Helvetica-Oblique
2	sans-serif	italic	bold	Helvetica-BoldOblique
3	monospace	normal	normal	Courier

Number	Font Family	Style	Weight	Font Name
3	monospace	normal	bold	Courier-Bold
3	monospace	italic	normal	Courier-Oblique
3	monospace	italic	bold	Courier-BoldOblique
4	Courier	normal	normal	Courier
4	Courier	normal	bold	Courier-Bold
4	Courier	italic	normal	Courier-Oblique
4	Courier	italic	bold	Courier-BoldOblique
5	Helvetica	normal	normal	Helvetica
5	Helvetica	normal	bold	Helvetica-Bold
5	Helvetica	italic	normal	Helvetica-Oblique
5	Helvetica	italic	bold	Helvetica-BoldOblique
6	Times	normal	normal	Times
6	Times	normal	bold	Times-Bold
6	Times	italic	normal	Times-Italic
6	Times	italic	bold	Times-BoldItalic
7	Symbol	normal	normal	Symbol
8	ZapfDingbats	normal	normal	ZapfDingbats

The TrueType fonts are listed in the following table. All TrueType fonts will be subsetted and embedded into PDF.

Number	Font Family Name	Style	Weight	Actual Font	Actual Font Type
1	Albany WT	normal	normal	ALBANYWT.ttf	TrueType (Latin1 only)
2	Albany WT J	normal	normal	ALBANWTJ.ttf	TrueType (Japanese flavor)
3	Albany WT K	normal	normal	ALBANWTK.ttf	TrueType (Korean flavor)
4	Albany WT SC	normal	normal	ALBANWTS.ttf	TrueType (Simplified Chinese flavor)
5	Albany WT TC	normal	normal	ALBANWTT.ttf	TrueType (Traditional Chinese flavor)
6	Andale Duospace WT	normal	normal	ADUO.ttf	TrueType (Latin1 only, Fixed width)
6	Andale Duospace WT	bold	bold	ADUOB.ttf	TrueType (Latin1 only, Fixed width)
7	Andale Duospace WT J	normal	normal	ADUOJ.ttf	TrueType (Japanese flavor, Fixed width)
7	Andale Duospace WT J	bold	bold	ADUOJB.ttf	TrueType (Japanese flavor, Fixed width)
8	Andale Duospace WT K	normal	normal	ADUOK.ttf	TrueType (Korean flavor, Fixed width)
8	Andale Duospace WT K	bold	bold	ADUOKB.ttf	TrueType (Korean flavor, Fixed width)

Number	Font Family Name	Style	Weight	Actual Font	Actual Font Type
9	Andale Duospace WT SC	normal	normal	ADUOSC.ttf	TrueType (Simplified Chinese flavor, Fixed width)
9	Andale Duospace WT SC	bold	bold	ADUOSCB.ttf	TrueType (Simplified Chinese flavor, Fixed width)
10	Andale Duospace WT TC	normal	normal	ADUOTC.ttf	TrueType (Traditional Chinese flavor, Fixed width)
10	Andale Duospace WT TC	bold	bold	ADUOTCB.ttf	TrueType (Traditional Chinese flavor, Fixed width)





---

# Using the XML Publisher APIs

## Introduction

This chapter is aimed at developers who wish to create programs or applications that interact with XML Publisher through its application programming interface. This information is meant to be used in conjunction with the Javadocs available from the Oracle Technology Network

[<http://www.oracle.com/technology/products/applications/publishing/index.html>].

This section assumes the reader is familiar with Java programming, XML, and XSL technologies.

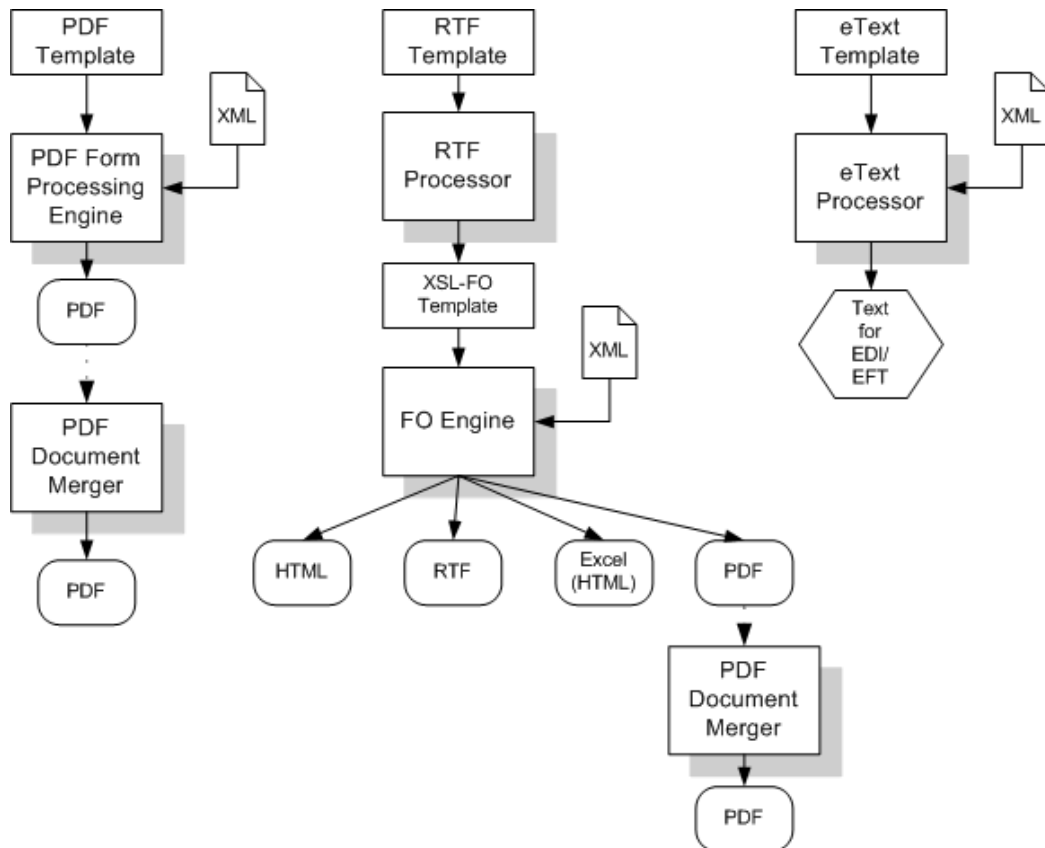
## XML Publisher Core APIs

XML Publisher is made up of the following core API components:

- **PDF Form Processing Engine**  
Merges a PDF template with XML data (and optional metadata) to produce PDF document output.
- **RTF Processor**  
Converts an RTF template to XSL in preparation for input to the FO Engine.
- **FO Engine**  
Merges XSL and XML to produce any of the following output formats: Excel (HTML), PDF, RTF, or HTML.
- **PDF Document Merger**  
Provides optional postprocessing of PDF files to merge documents, add page numbering, and set watermarks.

- eText Processor  
Converts RTF eText templates to XSL and merges the XSL with XML to produce text output for EDI and EFT transmissions.
- Document Processor (XML APIs)  
Provides batch processing functionality to access a single API or multiple APIs by passing a single XML file to specify template names, data sources, languages, output type, output names, and destinations.

The following diagram illustrates the template type and output type options for each core processing engine:



## PDF Form Processing Engine

The PDF Form Processing Engine creates a PDF document by merging a PDF template with an XML data file. This can be done using file names, streams, or an XML data string.

As input to the PDF Processing Engine you can optionally include an XML-based Template MetaInfo (.xtn) file. This is a supplemental template to define the placement of overflow data.

The FO Processing Engine also includes utilities to provide information about your PDF template. You can:

- Retrieve a list of field names from a PDF template
- Generate the XFDF data from the PDF template
- Convert XML data into XFDF using XSLT

## Merging a PDF Template with XML Data

XML data can be merged with a PDF template to produce a PDF output document in three ways:

- Using input/output file names
- Using input/output streams
- Using an input XML data string

You can optionally include a metadata XML file to describe the placement of overflow data in your template.

### Merging XML Data with a PDF Template Using Input/Output File Names

Input:

- Template file name (String)
- XML file name (String)
- Metadata XML file name (String)

Output:

- PDF file name (String)

#### Example

```
import oracle.apps.xdo.template.FormProcessor;
.
.
    FormProcessor fProcessor = new FormProcessor();

    fProcessor.setTemplate(args[0]); // Input File (PDF) name
    fProcessor.setData(args[1]);     // Input XML data file name
    fProcessor.setOutput(args[2]);   // Output File (PDF) name
    fProcessor.setMetaInfo(args[3]); // Metadata XML File name You
can omit this setting when you do not use Metadata.

    fProcessor.process();
```

## Merging XML Data with a PDF Template Using Input/Output Streams

Input:

- PDF Template (Input Stream)
- XML Data (Input Stream)
- Metadata XML Data (Input Stream)

Output:

- PDF (Output Stream)

### Example

```
import java.io.*;
import oracle.apps.xdo.template.FormProcessor;
.
.
.
    FormProcessor fProcessor = new FormProcessor();

    FileInputStream fIs = new FileInputStream(originalFilePath); // Input
File
    FileInputStream fIs2 = new FileInputStream(dataFilePath); // Input
Data
    FileInputStream fIs3 = new FileInputStream(metaData); // Metadata XML
Data
    FileOutputStream fOs = new FileOutputStream(newFilePath); // Output
File

    fProcessor.setTemplate(fIs);
    fProcessor.setData(fIs2); // Input Data
    fProcessor.setOutput(fOs);
    fProcessor.setMetaInfo(fIs3);
    fProcessor.process();

    fIs.close();
    fOs.close();
```

## Merging an XML Data String with a PDF Template

Input:

- Template file name (String)
- XML data (String)
- Metadata XML file name (String)

Output:

- PDF file name (String)

### Example

```
import oracle.apps.xdo.template.FormProcessor;
.
.
.
FormProcessor fProcessor = new FormProcessor();

fProcessor.setTemplate(originalFilePath);    // Input File (PDF) name
fProcessor.setDataString(xmlContents);       // Input XML string
fProcessor.setOutput(newFilePath);          // Output File (PDF) name
fProcessor.setMetaInfo(metaXml);            // Metadata XML File name    You
can omit this setting when you do not use Metadata.
fProcessor.process();
```

## Retrieving a List of Field Names

Use the `FormProcessor.getFieldNames()` API to retrieve the field names from a PDF template. The API returns the field names into an Enumeration object.

Input:

- PDF Template

Output:

- Enumeration Object

### Example

```
import java.util.Enumeration;
import oracle.apps.xdo.template.FormProcessor;
.
.
.
FormProcessor fProcessor = new FormProcessor();
fProcessor.setTemplate(filePath);           // Input File (PDF) name
Enumeration enum = fProcessor.getFieldNames();
while(enum.hasMoreElements()) {
    String formName = (String)enum.nextElement();
    System.out.println("name : " + formName + " , value : " +
fProcessor.getFieldValue(formName));
}
```

## Generating XFDF Data

XML Forms Data Format (XFDF) is a format for representing forms data and annotations in a PDF document. XFDF is the XML version of Forms Data Format (FDF), a simplified version of PDF for representing forms data and annotations. Form fields in a PDF document include edit boxes, buttons, and radio buttons.

Use this class to generate XFDF data. When you create an instance of this class, an internal XFDF tree is initialized. Use `append()` methods to append a FIELD element to the XFDF tree by passing a String name-value pair. You can append data as many times as you want.

This class also allows you to append XML data by calling `appendXML()` methods. Note that you must set the appropriate XSL stylesheet by calling `setStyleSheet()` method before calling `appendXML()` methods. You can append XML data as many times as you want.

You can retrieve the internal XFDF document at any time by calling one of the following methods: `toString()`, `toReader()`, `toInputStream()`, or `toXMLDocument()`.

The following is a sample of XFDF data:

#### Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xfdf xmlns="http://ns.adobe.com/xfdf/" xml:space="preserve">
<fields>
  <field name="TITLE">
    <value>Purchase Order</value>
  </field>
  <field name="SUPPLIER_TITLE">
    <value>Supplier</value>
  </field>
  ...
</fields>
```

The following code example shows how the API can be used:

#### Example

```
import oracle.apps.xdo.template.FormProcessor;
import oracle.apps.xdo.template.pdf.xfdf.XFDFObject;
.
.
.
FormProcessor fProcessor = new FormProcessor();
fProcessor.setTemplate(filePath); // Input File (PDF) name
XFDFObject xfdfObject = new XFDFObject(fProcessor.getFieldInfo());
System.out.println(xfdfObject.toString());
```

## Converting XML Data into XFDF Format Using XSLT

Use an XSL stylesheet to convert standard XML to the XFDF format. Following is an example of the conversion of sample XML data to XFDF:

Assume your starting XML has a ROWSET/ROW format as follows:

```
<ROWSET>
  <ROW num="0">
    <SUPPLIER>Supplier</SUPPLIER>
    <SUPPLIERNUMBER>Supplier Number</SUPPLIERNUMBER>
    <CURRCODE>Currency</CURRCODE>
  </ROW>
  ...
</ROWSET>
```

From this XML you want to generate the following XFDF format:

```

<fields>
  <field name="SUPPLIER1">
    <value>Supplier</value>
  </field>
  <field name="SUPPLIERNUMBER1">
    <value>Supplier Number</value>
  </field>
  <field name="CURRCODE1">
    <value>Currency</value>
  </field>
  ...
</fields>

```

The following XSLT will carry out the transformation:

```

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<fields>
<xsl:apply-templates/>
</fields>
</xsl:template>
  <!-- Count how many ROWs(rows) are in the source XML file. -->
  <xsl:variable name="cnt" select="count(//row//ROW)" />
  <!-- Try to match ROW (or row) element.
  <xsl:template match="ROW/*|row/*">
    <field>
      <!-- Set "name" attribute in "field" element. -->
      <xsl:attribute name="name">
        <!-- Set the name of the current element (column name) as a
value of the current name attribute. -->
        <xsl:value-of select="name(.)" />
        <!-- Add the number at the end of the name attribute value
if more than 1 rows found in the source XML file.-->
        <xsl:if test="$cnt > 1">
          <xsl:number count="ROW|row" level="single" format="1"/>
        </xsl:if>
      </xsl:attribute>
      <value>
        <!--Set the text data set in the current column data as a
text of the "value" element. -->
        <xsl:value-of select="." />
      </value>
    </field>
  </xsl:template>
</xsl:stylesheet>

```

You can then use the XFDFObject to convert XML to the XFDF format using an XSLT as follows:

### Example

```
import java.io.*;
import oracle.apps.xdo.template.pdf.xfdf.XFDFObject;
.
.
.
XFDFObject xfdfObject = new XFDFObject();

xfdfObject .setStylesheet(new BufferedInputStream(new
FileInputStream(xslPath))); // XSL file name
xfdfObject .appendXML( new File(xmlPath1)); // XML data file name
xfdfObject .appendXML( new File(xmlPath2)); // XML data file name

System.out.print(xfdfObject .toString());
```

## RTF Processor Engine

### Generating XSL

The RTF processor engine takes an RTF template as input. The processor parses the template and creates an XSL-FO template. This can then be passed along with a data source (XML file) to the FO Engine to produce PDF, HTML, RTF, or Excel (HTML) output.

Use either input/output file names or input/output streams as shown in the following examples:

#### Generating XSL with Input/Output File Names

Input:

- RTF file name (String)

Output:

- XSL file name (String)

### Example

```
import oracle.apps.xdo.template.FOProcessor;
.
.
.
    public static void main(String[] args)
    {
RTFProcessor rtfProcessor = new RTFProcessor(args[0]); //input template
rtfProcessor.setOutput(args[1]); // output file
rtfProcessor.process();
        System.exit(0);
    }
```

#### Generating XSL with Input/Output Stream

Input:



- RTF (InputStream)

Output:

- XSL (OutputStream)

### Example

```
import oracle.apps.xdo.template.FOProcessor;
.
.
.
public static void main(String[] args)
{
    FileInputStream fIs = new FileInputStream(args[0]); //input
template
    FileOutputStream fOs = new FileOutputStream(args[1]); // output

    RTFProcessor rtfProcessor = new RTFProcessor(fIs);
    rtfProcessor.setOutput(fOs);
    rtfProcessor.process();
    // Closes inputStreams outputStream
    System.exit(0);
}
```

## FO Processor Engine

### Generating Output from an XML File and an XSL File

The FO Processor Engine is XML Publisher's implementation of the W3C XSL-FO standard. It does not represent a complete implementation of every XSL-FO component. For a list of supported XSL-FO elements, see Supported XSL-FO Elements, page A-1.

The FO Processor can generate output in PDF, RTF, HTML, or Excel (HTML) from either of the following two inputs:

- Template (XSL) and Data (XML) combination
- FO object

Both input types can be passed as file names, streams, or in an array. Set the output format by setting the `setOutputFormat` method to one of the following:

- `FORMAT_EXCEL`
- `FORMAT_HTML`
- `FORMAT_PDF`
- `FORMAT_RTF`

An XSL-FO utility is also provided that creates XSL-FO from the following inputs:

- XSL file and XML file
- Two XML files and two XSL files
- Two XSL-FO files (merge)

The FO object output from the XSL-FO utility can then be used as input to the FO processor.

## Major Features of the FO Processor

### **Bidirectional Text**

XML Publisher utilizes the Unicode BiDi algorithm for BiDi layout. Based on specific values for the properties writing-mode, direction, and unicode bidi, the FO Processor supports the BiDi layout.

The writing-mode property defines how word order is supported in lines and order of lines in text. That is: right-to-left, top-to-bottom or left-to-right, top-to-bottom. The direction property determines how a string of text will be written: that is, in a specific direction, such as right-to-left or left-to-right. The unicode bidi controls and manages override behavior.

### **Font Fallback Mechanism**

The FO Processor supports a two-level font fallback mechanism. This mechanism provides control over what default fonts to use when a specified font or glyph is not found. XML Publisher provides appropriate default fallback fonts automatically without requiring any configuration. XML Publisher also supports user-defined configuration files that specify the default fonts to use. For glyph fallback, the default mechanism will only replace the glyph and not the entire string.

### **Variable Header and Footer**

For headers and footers that require more space than what is defined in the template, the FO Processor extends the regions and reduces the body region by the difference between the value of the page header and footer and the value of the body region margin.

### **Horizontal Table Break**

This feature supports a "Z style" of horizontal table break. The horizontal table break is not sensitive to column span, so that if the column-spanned cells exceed the page (or area width), the FO Processor splits it and does not apply any intelligent formatting to the split cell.

The following figure shows a table that is too wide to display on one page:

Title				
1	2	3	4	5

Page Number

The following figure shows one option of how the horizontal table break will handle the wide table. In this example, a horizontal table break is inserted after the third column.

Title		
1	2	3

1

Title	
4	5

2

The following figure shows another option. The table breaks after the third column, but includes the first column with each new page.

Title		
1	2	3
1		

Title	
1	4
2	

Title	
1	5
3	

### Generating Output Using File Names

The following example shows how to use the FO Processor to create an output file using file names.

Input:

- XML file name (String)
- XSL file name (String)

Output:

- Output file name (String)

#### Example

```
import oracle.apps.xdo.template.FOProcessor;
.
.
.
public static void main(String[] args)
{
    FOProcessor processor = new FOProcessor();
    processor.setData(args[0]);    // set XML input file
    processor.setTemplate(args[1]); // set XSL input file
    processor.setOutput(args[2]);  //set output file
    processor.setOutputFormat(FOProcessor.FORMAT_PDF);
    // Start processing
    try
    {
        processor.generate();
    }
    catch (XDOException e)
    {
        e.printStackTrace();
        System.exit(1);
    }

    System.exit(0);
}
```

## Generating Output Using Streams

The processor can also be used with input/output streams as shown in the following example:

Input:

- XML data (InputStream)
- XSL data (InputStream)

Output:

- Output stream (OutputStream)

### Example

```
import java.io.InputStream;
import java.io.OutputStream;
import oracle.apps.xdo.template.FOProcessor;
.
.
.
public void runFOProcessor(InputStream xmlInputStream,
                           InputStream xslInputStream,
                           OutputStream pdfOutputStream)
{
    FOProcessor processor = new FOProcessor();
    processor.setData(xmlInputStream);
    processor.setTemplate(xslInputStream);
    processor.setOutput(pdfOutputStream);
    // Set output format (for PDF generation)
    processor.setOutputFormat(FOProcessor.FORMAT_PDF);
    // Start processing
    try
    {
        processor.generate();
    }
    catch (XDOException e)
    {
        e.printStackTrace();
        System.exit(1);
    }

    System.exit(0);
}
```

## Generating Output from an Array of XSL Templates and XML Data

An array of data and template combinations can be processed to generate a single output file from the multiple inputs. The number of input data sources must match the number of templates that are to be applied to the data. For example, an input of File1.xml, File2.xml, File3.xml and File1.xsl, File2.xsl, and File3.xsl will produce a single File1\_File2\_File3.pdf.

Input:

- XML data (Array)
- XSL data (template) (Array)

Output:

- File Name (String)

### Example

```
import java.io.InputStream;
import java.io.OutputStream;
import oracle.apps.xdo.template.FOProcessor;
.
.
.
    public static void main(String[] args)
    {

        String[] xmlInput = {"first.xml", "second.xml", "third.xml"};
        String[] xslInput = {"first.xsl", "second.xsl", "third.xsl"};

        FOProcessor processor = new FOProcessor();
        processor.setData(xmlInput);
        processor.setTemplate(xslInput);

        processor.setOutput("/tmp/output.pdf");           //set (PDF) output
file
        processor.setOutputFormat(FOProcessor.FORMAT_PDF);
        processor.process();
        // Start processing
        try
        {
            processor.generate();
        }
        catch (XDOException e)
        {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

## Using the XSL-FO Utility

Use the XSL-FO Utility to create an XSL-FO output file from input XML and XSL files, or to merge two XSL-FO files. Output from this utility can be used to generate your final output. See *Generating Output from an XSL-FO file*, page 8-17.

### Creating XSL-FO from an XML File and an XSL File

Input:

- XML file

- XSL file

Output:

- XSL-FO (InputStream)

### Example

```
import oracle.apps.xdo.template.fo.util.FOUtility;
.
.
.
public static void main(String[] args)
{
    InputStream foStream;

    // creates XSL-FO InputStream from XML(arg[0])
    // and XSL(arg[1]) filepath String
    foStream = FOUtility.createFO(args[0], args[1]);
    if (mergedFOStream == null)
    {
        System.out.println("Merge failed.");
        System.exit(1);
    }

    System.exit(0);
}
```

### Creating XSL-FO from Two XML Files and Two XSL files

Input:

- XML File 1
- XML File 2
- XSL File 1
- XSL File 2

Output:

- XSL-FO (InputStream)

### Example

```
import oracle.apps.xdo.template.fo.util.FOUtility;
.
.
.
public static void main(String[] args)
{
    InputStream firstFOStream, secondFOStream, mergedFOStream;
    InputStream[] input = new InputStream[2];

    // creates XSL-FO from arguments
    firstFOStream = FOUtility.createFO(args[0], args[1]);

    // creates another XSL-FO from arguments
    secondFOStream = FOUtility.createFO(args[2], args[3]);

    // set each InputStream into the InputStream Array
    Array.set(input, 0, firstFOStream);
    Array.set(input, 1, secondFOStream);

    // merges two XSL-FOs
    mergedFOStream = FOUtility.mergeFOs(input);

    if (mergedFOStream == null)
    {
        System.out.println("Merge failed.");
        System.exit(1);
    }
    System.exit(0);
}
```

### Merging Two XSL-FO Files

Input:

- Two XSL-FO file names (Array)

Output:

- One XSL-FO (InputStream)



### Example

```
import oracle.apps.xdo.template.fo.util.FOUtility;
.
.
.
public static void main(String[] args)
{
    InputStream mergedFOStream;

    // creates Array
    String[] input = {args[0], args[1]};

    // merges two FO files
    mergedFOStream = FOUtility.mergeFOs(input);
    if (mergedFOStream == null)
    {
        System.out.println("Merge failed.");
        System.exit(1);
    }
    System.exit(0);
}
```

## Generating Output from an FO file

The FO Processor can also be used to process an FO object to generate your final output. An FO object is the result of the application of an XSL-FO stylesheet to XML data. These objects can be generated from a third party application and fed as input to the FO Processor.

The processor is called using a similar method to those already described, but a template is not required as the formatting instructions are contained in the FO.

### Generating Output Using File Names

Input:

- FO file name (String)

Output:

- PDF file name (String)

### Example

```
import oracle.apps.xdo.template.FOProcessor;
.
.
.
public static void main(String[] args) {

    FOProcessor processor = new FOProcessor();
    processor.setData(args[0]);    // set XSL-FO input file
    processor.setTemplate((String)null);
    processor.setOutput(args[2]); //set (PDF) output file
    processor.setOutputFormat(FOProcessor.FORMAT_PDF);
    // Start processing
    try
    {
        processor.generate();
    }
    catch (XDOException e)
    {
        e.printStackTrace();
        System.exit(1);
    }

    System.exit(0);
}
```

### Generating Output Using Streams

Input:

- FO data (InputStream)

Output:

- Output (OutputStream)

### Example

```
import java.io.InputStream;
import java.io.OutputStream;
import oracle.apps.xdo.template.FOProcessor;
.
.
.
public void runFOProcessor(InputStream xmlfoInputStream,
                           OutputStream pdfOutputStream)
{
    FOProcessor processor = new FOProcessor();
    processor.setData(xmlfoInputStream);
    processor.setTemplate((String)null);

    processor.setOutput(pdfOutputStream);
    // Set output format (for PDF generation)
    processor.setOutputFormat(FOProcessor.FORMAT_PDF);
    // Start processing
    try
    {
        processor.generate();
    }
    catch (XDOException e)
    {
        e.printStackTrace();
        System.exit(1);
    }
}
```

### Generating Output with an Array of FO Data

Pass multiple FO inputs as an array to generate a single output file. A template is not required, therefore set the members of the template array to null, as shown in the example.

Input:

- FO data (Array)

Output:

- Output File Name (String)

### Example

```
import java.lang.reflect.Array;
import oracle.apps.xdo.template.FOProcessor;
.
.
.
    public static void main(String[] args)
    {

        String[] xmlInput = {"first.fo", "second.fo", "third.fo"};
        String[] xslInput = {null, null, null};    // null needs for xsl-fo
        input

        FOProcessor processor = new FOProcessor();
        processor.setData(xmlInput);
        processor.setTemplate(xslInput);

        processor.setOutput("/tmp/output.pdf);          //set (PDF) output
        file
        processor.setOutputFormat(FOProcessor.FORMAT_PDF);
        processor.process();
        // Start processing
        try
        {
            processor.generate();
        }
        catch (XDOException e)
        {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

## PDF Document Merger

The PDF Document Merger class provides a set of utilities to manipulate PDF documents. Using these utilities, you can merge documents, add page numbering, set backgrounds, and add watermarks.

### Merging PDF Documents

Many business documents are composed of several individual documents that need to be merged into a single final document. The PDFDocMerger class supports the merging of multiple documents to create a single PDF document. This can then be manipulated further to add page numbering, watermarks, or other background images.

#### Merging with Input/Output File Names

The following code demonstrates how to merge (concatenate) two PDF documents using physical files to generate a single output document.

Input:

- PDF\_1 file name (String)
- PDF\_2 file name (String)

Output:

- PDF file name (String)

### Example

```
import java.io.*;
import oracle.apps.xdo.common.pdf.util.PDFDocMerger;
.
.
.
public static void main(String[] args)
{
    try
    {
        // Last argument is PDF file name for output
        int inputNumbers = args.length - 1;

        // Initialize inputStreams
        FileInputStream[] inputStreams = new
FileInputStream[inputNumbers];
        inputStreams[0] = new FileInputStream(args[0]);
        inputStreams[1] = new FileInputStream(args[1]);

        // Initialize outputStream
        FileOutputStream outputStream = new FileOutputStream(args[2]);

        // Initialize PDFDocMerger
        PDFDocMerger docMerger = new PDFDocMerger(inputStreams,
outputStream);

        // Merge PDF Documents and generates new PDF Document
        docMerger.mergePDFDocs();
        docMerger = null;

        // Closes inputStreams and outputStream
    }
    catch(Exception exc)
    {
        exc.printStackTrace();
    }
}
```

### Merging with Input/Output Streams

Input:

- PDF Documents (InputStream Array)

Output:

- PDF Document (OutputStream)

### Example

```
import java.io.*;
import oracle.apps.xdo.common.pdf.util.PDFDocMerger;
.
.
.
    public boolean mergeDocs(InputStream[] inputStreams, OutputStream
outputStream)
    {
        try
        {
            // Initialize PDFDocMerger
            PDFDocMerger docMerger = new PDFDocMerger(inputStreams,
outputStream);

            // Merge PDF Documents and generates new PDF Document
            docMerger.mergePDFDocs();
            docMerger = null;

            return true;
        }
        catch(Exception exc)
        {
            exc.printStackTrace();
            return false;
        }
    }
}
```

### Merging with Background to Place Page Numbering

The following code demonstrates how to merge two PDF documents using input streams to generate a single merged output stream.

To add page numbers:

1. Create a background PDF template document that includes a PDF form field in the position that you would like the page number to appear on the final output PDF document.
2. Name the form field @pagenum@.
3. Enter the number in the field from which to start the page numbering. If you do not enter a value in the field, the start page number defaults to 1.

Input:

- PDF Documents (InputStream Array)
- Background PDF Document (InputStream)

Output:

- PDF Document (OutputStream)

### Example

```
import java.io.*;
import oracle.apps.xdo.common.pdf.util.PDFDocMerger;
.
.
.
public static boolean mergeDocs(InputStream[] inputStreams, InputStream
backgroundStream, OutputStream outputStream)

{
    try
    {
        // Initialize PDFDocMerger
        PDFDocMerger docMerger = new PDFDocMerger(inputStreams,
outputStream);

        // Set Background
        docMerger.setBackground(backgroundStream);

        // Merge PDF Documents and generates new PDF Document
        docMerger.mergePDFDocs();
        docMerger = null;

        return true;
    }
    catch(Exception exc)
    {
        exc.printStackTrace();
        return false;
    }
}
```

### Adding Page Numbers to Merged Documents

The FO Processor supports page numbering natively through the XSL-FO templates, but if you are merging multiple documents you must use this class to number the complete document from beginning to end.

The following code example places page numbers in a specific point on the page, formats the numbers, and sets the start value using the following methods:

- `setPageNumberCoordinates (x, y)` - sets the x and y coordinates for the page number position. The following example sets the coordinates to 300, 20.
- `setPageNumberFontInfo (font name, size)` - sets the font and size for the page number. If you do not call this method, the default "Helvetica", size 8 is used. The following example sets the font to "Courier", size 8.
- `setPageNumberValue (n, n)` - sets the start number and the page on which to begin numbering. If you do not call this method, the default values 1, 1 are used.

Input:

- PDF Documents (InputStream Array)

Output:

- PDF Document (OutputStream)

### Example

```
import java.io.*;
import oracle.apps.xdo.common.pdf.util.PDFDocMerger;
.
.
.
    public boolean mergeDocs(InputStream[] inputStreams, OutputStream
outputStream)
    {
        try
        {
            // Initialize PDFDocMerger
            PDFDocMerger docMerger = new PDFDocMerger(inputStreams,
outputStream);

            // Calls several methods to specify Page Number

            // Calling setPageNumberCoordinates() method is necessary to set
Page Numbering
            // Please refer to javadoc for more information
            docMerger.setPageNumberCoordinates(300, 20);

            // If this method is not called, then the default font"(Helvetica,
8)" is used.
            docMerger.setPageNumberFontInfo("Courier", 8);

            // If this method is not called, then the default initial value
"(1, 1)" is used.
            docMerger.setPageNumberValue(1, 1);

            // Merge PDF Documents and generates new PDF Document
            docMerger.mergePDFDocs();
            docMerger = null;

            return true;
        }
        catch(Exception exc)
        {
            exc.printStackTrace();
            return false;
        }
    }
}
```

## Setting a Text or Image Watermark

Some documents that are in a draft phase require that a watermark indicating "DRAFT" be displayed throughout the document. Other documents might require a background image on the document. The following code sample shows how to use the PDFDocMerger class to set a watermark.

### Setting a Text Watermark

Use the SetTextDefaultWatermark( ) method to set a text watermark with the following attributes:



- Text angle (in degrees): 55
- Color: light gray (0.9, 0.9, 0.9)
- Font: Helvetica
- Font Size: 100
- The start position is calculated based on the length of the text

Alternatively, use the `SetTextWatermark( )` method to set each attribute separately. Use the `SetTextWatermark()` method as follows:

- `SetTextWatermark ("Watermark Text", x, y)` - declare the watermark text, and set the x and y coordinates of the start position. In the following example, the watermark text is "Draft" and the coordinates are 200f, 200f.
- `setTextWatermarkAngle (n)` - sets the angle of the watermark text. If this method is not called, 0 will be used.
- `setTextWatermarkColor (R, G, B)` - sets the RGB color. If this method is not called, light gray (0.9, 0.9, 0.9) will be used.
- `setTextWatermarkFont ("font name", font size)` - sets the font and size. If you do not call this method, Helvetica, 100 will be used.

The following example shows how to set these properties and then call the `PDFDocMerger`.

Input:

- PDF Documents (InputStream)

Output:

- PDF Document (OutputStream)

### Example

```
import java.io.*;
import oracle.apps.xdo.common.pdf.util.PDFDocMerger;
.
.
.
    public boolean mergeDocs(InputStream inputStreams, OutputStream
outputStream)
    {
        try
        {
            // Initialize PDFDocMerger
            PDFDocMerger docMerger = new PDFDocMerger(inputStreams,
outputStream);

            // You can use setTextDefaultWatermark() without these detailed
setting
            docMerger.setTextWatermark("DRAFT", 200f, 200f); //set text and
place
            docMerger.setTextWatermarkAngle(80);             //set angle
            docMerger.setTextWatermarkColor(1.0f, 0.3f, 0.5f); // set RGB
Color

            // Merge PDF Documents and generates new PDF Document
            docMerger.mergePDFDocs();
            docMerger = null;

            return true;
        }
        catch(Exception exc)
        {
            exc.printStackTrace();
            return false;
        }
    }
}
```

### Setting Image Watermark

An image watermark can be set to cover the entire background of a document, or just to cover a specific area (for example, to display a logo). Specify the placement and size of the image using rectangular coordinates as follows:

```
float[ ] rct = {LowerLeft X, LowerLeft Y, UpperRight X,
UpperRight Y}
```

For example:

```
float[ ] rct = {100f, 100f, 200f, 200f}
```

The image will be sized to fit the rectangular area defined.

To use the actual image size, without sizing it, define the LowerLeft X and LowerLeft Y positions to define the placement and specify the UpperRight X and UpperRight Y coordinates as -1f. For example:

```
float[ ] rct = {100f, 100f, -1f, -1f}
```

Input:

- PDF Documents (InputStream)
- Image File (InputStream)

Output:

- PDF Document (OutputStream)

### Example

```
import java.io.*;
import oracle.apps.xdo.common.pdf.util.PDFDocMerger;
.
.
.
    public boolean mergeDocs(InputStream inputStreams, OutputStream
outputStream, String imagePath)
    {
        try
        {
            // Initialize PDFDocMerger
            PDFDocMerger docMerger = new PDFDocMerger(inputStreams,
outputStream);

            FileInputStream wmStream = new FileInputStream(imageFilePath);
            float[] rct = {100f, 100f, -1f, -1f};
            pdfMerger.setImageWatermark(wmStream, rct);

            // Merge PDF Documents and generates new PDF Document
            docMerger.mergePDFDocs();
            docMerger = null;

            // Closes inputStreams
            return true;
        }
        catch(Exception exc)
        {
            exc.printStackTrace();
            return false;
        }
    }
}
```

## PDF Book Binder Processor

The PDFBookBinder processor is useful for the merging of multiple PDF documents into a single document consisting of a hierarchy of chapters, sections, and subsections and a table of contents for the document. The processor also generates PDF style "bookmarks"; the outline structure is determined by the chapter and section hierarchy. The processor is extremely powerful allowing you complete control over the combined document.

### Usage

The table of contents formatting and style is created through the use of an RTF template created in Microsoft Word. The chapters are passed into the program as separate PDF

files (one chapter, section, or subsection corresponds to one PDF file). Templates may also be specified at the chapter level for insertion of dynamic or static content, page numbering, and placement of hyperlinks within the document.

The templates can be in RTF or PDF format. RTF templates are more flexible by allowing you to leverage XML Publisher's support for dynamic content. PDF templates are much less flexible, making it difficult to achieve desirable effects such as the reflow of text areas when inserting page numbers and other types of dynamic content.

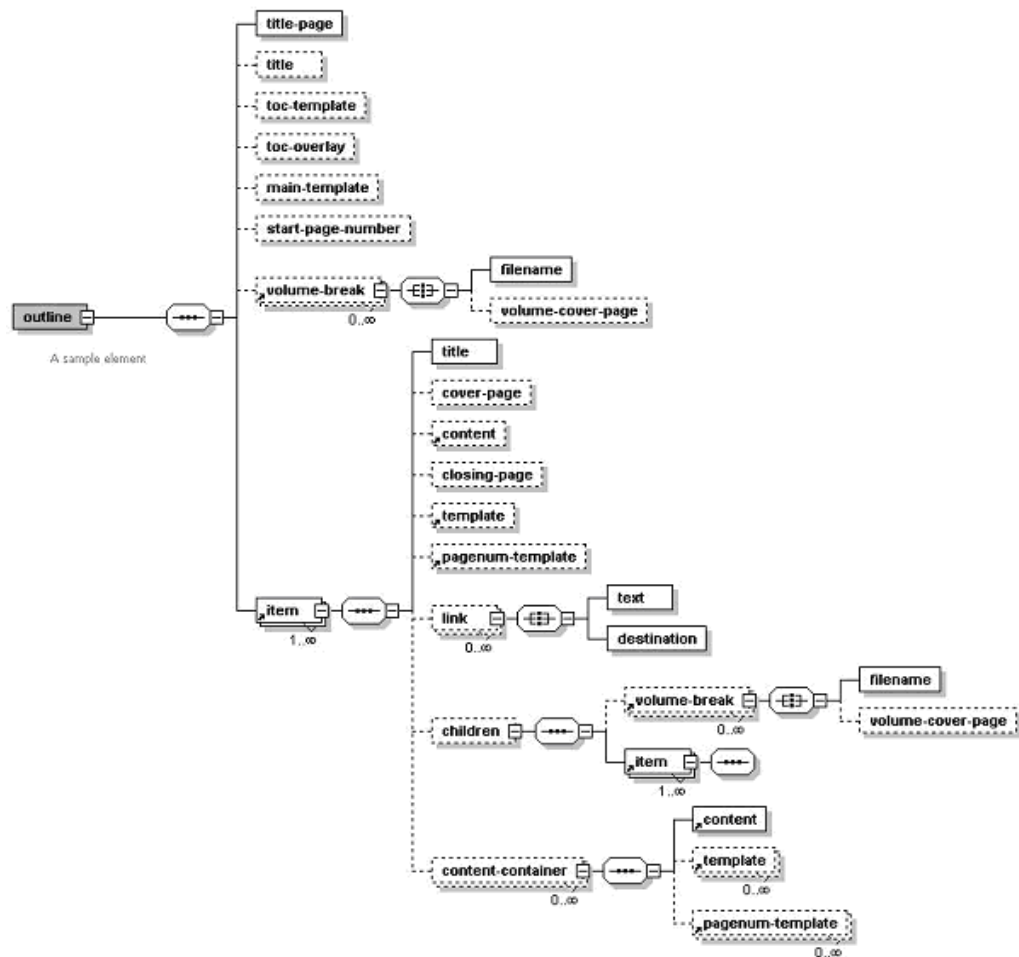
The templates can be rotated (at right angles) or be made transparent. A PDF template can also be specified at the book level, enabling the ability to specify global page numbering, or other content such as backgrounds and watermarks. A title page can also be passed in as a parameter, as well as cover and closing pages for each chapter or section.

## **XML Control File**

The structure of the book's chapters, sections, and subsections is represented as XML and passed in as a command line parameter; or it can also be passed in at the API level. All of the chapter and section files, as well as all the templates files and their respective parameters, are specified inside this XML structure. Therefore, the only two required parameters are an XML file and a PDF output file.

You can also specify volume breaks inside the book structure. Specifying volume breaks will split the content up into separate output files for easier file and printer management.

The structure of the XML control file is represented in the following diagram:



To specify template and content file locations in your XML structure, you can specify a path relative to your local file system or you can specify a URL referring to the template or content location. Secure HTTP protocol is supported, as well as the following XML Publisher protocol:

- "blob://" - used for specifying data in any user-defined BLOB table.

The format for the "blob://" protocol is:

```
blob://[table_name].[blob_column_name]/[pk_datatype]:[pk_name]=[pk_value]/.../...
```

## Command Line Options

Following is an example of the command line usage:

```
java oracle.apps.template.pdf.book.PDFBookBinder [-debug <true or false>] [-tmp <temp dir>] -xml <input xml> -pdf <output pdf>
```

where

-xml <file> is the file name of the input XML file containing the table of contents XML structure.

-pdf <file> is the final generated PDF output file.

-tmp <directory> is the temporary directory for better memory management. (This is optional, if not specified, the system environment variable "java.io.tmpdir" will be used.)

-log <file> sets the output log file (optional, default is System.out).

-debug <true or false> turns debugging off or on.

## API Method Call

The following is an example of an API method call:

```
String xmlInputPath = "c:\\tmp\\toc.xml";
String pdfOutputPath = "c:\\tmp\\final_book.pdf";
PDFBookBinder bookBinder = new PDFBookBinder(xmlInputPath,
    pdfOutputPath);

bookBinder.setConfig(new Properties());
bookBinder.process();
```

## Document Processor Engine

The Document Processor Engine provides batch processing functionality to access a single API or multiple APIs by passing a single XML instance document to specify template names, data sources, languages, output type, output names, and destinations.

This solution enables batch printing with XML Publisher, in which a single XML document can be used to define a set of invoices for customers, including the preferred output format and delivery channel for those customers. The XML format is very flexible allowing multiple documents to be created or a single master document.

This section:

- Describes the hierarchy and elements of the Document Processor XML file
- Provides sample XML files to demonstrate specific processing options
- Provides example code to invoke the processors

## Hierarchy and Elements of the Document Processor XML File

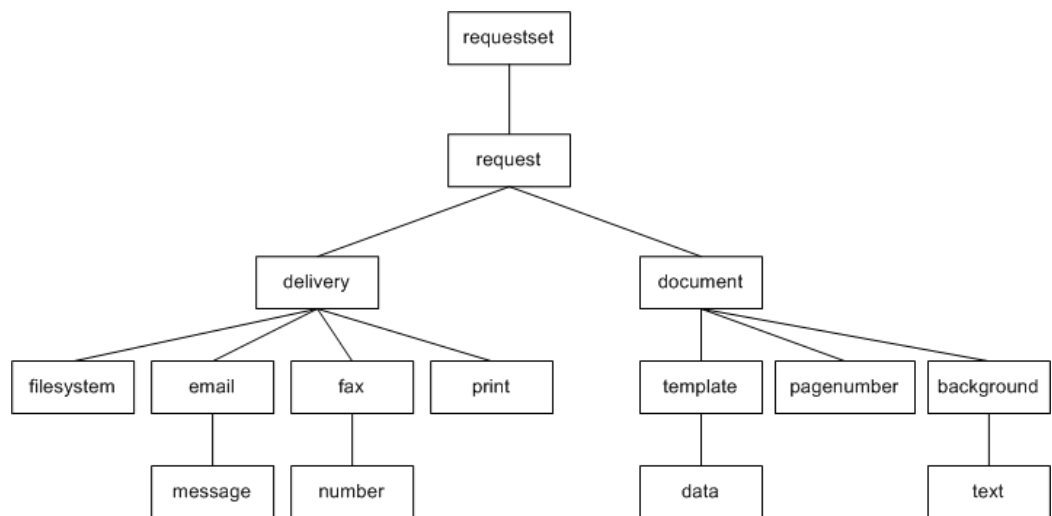
The Document Processor XML file has the following element hierarchy:

```

Requestset
  request
    delivery
      filesystem
      print
      fax
        number
      email
        message
    document
      background
        text
      pagenumber
      template
      data

```

This hierarchy is displayed in the following illustration:



The following table describes each of the elements:

Element	Attributes	Description
requestset	xmlns version	Root element must contain [ xmlns:xapi="http://xmlns.oracle.com/oxp/xapi/" ] block  The version is not required, but defaults to "1.0".
request	N/A	Element that contains the data and template processing definitions.

Element	Attributes	Description
delivery	N/A	Defines where the generated output is sent.
document	output-type	Specify one output that can have several template elements. The output-type attribute is optional. Valid values are:  pdf (Default)  rtf  html  excel  text
filesystem	output	Specify this element to save the output to the file system. Define the directory path in the output attribute.
print	<ul style="list-style-type: none"> <li>printer</li> <li>server-alias</li> </ul>	The print element can occur multiple times under delivery to print one document to several printers. Specify the printer attribute as a URI, such as: "ipp://myprintserver:631/printers/printrname"
fax	<ul style="list-style-type: none"> <li>server</li> <li>server-alias</li> </ul>	Specify a URI in the server attribute, for example: "ipp://myfaxserver1:631/printers/myfaxmachine"
number		The number element can occur multiple times to list multiple fax numbers. Each element occurrence must contain only one number.



Element	Attributes	Description
email	<ul style="list-style-type: none"> <li>server</li> <li>port</li> <li>from</li> <li>reply-to</li> <li>server-alias</li> </ul>	<p>Specify the outgoing mail server (SMTP) in the <code>server</code> attribute.</p> <p>Specify the mail server port in the <code>port</code> attribute.</p>
message	<ul style="list-style-type: none"> <li>to</li> <li>cc</li> <li>bcc</li> <li>attachment</li> <li>subject</li> </ul>	<p>The <code>message</code> element can be placed several times under the <code>email</code> element. You can specify character data in the <code>message</code> element.</p> <p>You can specify multiple e-mail addresses in the <code>to</code>, <code>cc</code> and <code>bcc</code> attributes separated by a comma.</p> <p>The <code>attachment</code> value is either <code>true</code> or <code>false</code> (default). If <code>attachment</code> is <code>true</code>, then a generated document will be attached when the e-mail is sent.</p> <p>The <code>subject</code> attribute is optional.</p>
background	where	<p>If the background text is required on a specific page, then set the <code>where</code> value to the page numbers required. The page index starts at 1. The default value is 0, which places the background on all pages.</p>

Element	Attributes	Description
text	<ul style="list-style-type: none"> <li>title</li> <li>default</li> </ul>	<p>Specify the watermark text in the <code>title</code> value.</p> <p>A default value of "yes" automatically draws the watermark with forward slash type. The default value is yes.</p>
pagenumber	<ul style="list-style-type: none"> <li>initial-page-index</li> <li>initial-value</li> <li>x-pos</li> <li>y-pos</li> </ul>	<p>The <code>initial-page-index</code> default value is 0.</p> <p>The <code>initial-value</code> default value is 1.</p> <p>"Helvetica" is used for the page number font.</p> <p>The <code>x-pos</code> provides lower left x position.</p> <p>The <code>y-pos</code> provides lower left y position.</p>
template	<ul style="list-style-type: none"> <li>locale</li> <li>location</li> <li>type</li> </ul>	<p>Contains template information.</p> <p>Valid values for the <code>type</code> attribute are</p> <p>pdf</p> <p>rtf</p> <p>xsl-fo</p> <p>etext</p> <p>The default value is "pdf".</p>

Element	Attributes	Description
data	location	<p>Define the <code>location</code> attribute to specify the location of the data, or attach the actual XML data with subelements. The default value of <code>location</code> is "inline". If the <code>location</code> points to either an XML file or a URL, then the data should contain an XML declaration with the proper encoding.</p> <p>If the <code>location</code> attribute is not specified, the <code>data</code> element should contain the subelements for the actual data. This must not include an XML declaration.</p>

## XML File Samples

Following are sample XML files that show:

- Simple XML shape
- Defining two data sets
- Defining multiple templates and data
- Retrieving templates over HTTP
- Retrieving data over HTTP
- Generating more than one output
- Defining page numbers

### Simple XML sample

The following sample is a simple example that shows the definition of one template (`template1.pdf`) and one data source (`data1`) to produce one output file (`outfile.pdf`) delivered to the file system:

### Example

```
<?xml version="1.0" encoding="UTF-8" ?>
  <xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
    <xapi:request>
      <xapi:delivery>
        <xapi:filesystem output="d:\tmp\outfile.pdf" />
      </xapi:delivery>
      <xapi:document output-type="pdf">
        <xapi:template type="pdf" location="d:\mywork\template1.pdf">
          <xapi:data>
            <field1>data1</field1>
          </xapi:data>
        </xapi:template>
      </xapi:document>
    </xapi:request>
  </xapi:requestset>
```

### Defining two data sets

The following example shows how to define two data sources to merge with one template to produce one output file delivered to the file system:

### Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
  <xapi:request>
    <xapi:delivery>
      <xapi:filesystem output="d:\tmp\outfile.pdf"/>
    </xapi:delivery>

    <xapi:document output-type="pdf">
      <xapi:template type="pdf"
        location="d:\mywork\template1.pdf">
        <xapi:data>
          <field1>The first set of data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The second set of data</field1>
        </xapi:data>
      </xapi:template>
    </xapi:document>
  </xapi:request>
</xapi:requestset>
```

### Defining multiple templates and data

The following example builds on the previous examples by applying two data sources to one template and two data sources to a second template, and then merging the two into a single output file. Note that when merging documents, the `output-type` must be "pdf".

### Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
  <xapi:request>
    <xapi:delivery>
      <xapi:filesystem output="d:\tmp\outfile3.pdf"/>
    </xapi:delivery>

    <xapi:document output-type="pdf">
      <xapi:template type="pdf"
        location="d:\mywork\template1.pdf">
        <xapi:data>
          <field1>The first set of data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The second set of data</field1>
        </xapi:data>
      </xapi:template>

      <xapi:template type="pdf"
        location="d:\mywork\template2.pdf">
        <xapi:data>
          <field1>The third set of data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The fourth set of data</field1>
        </xapi:data>
      </xapi:template>
    </xapi:document>
  </xapi:request>
</xapi:requestset>
```

### Retrieving templates over HTTP

This sample is identical to the previous example, except in this case the two templates are retrieved over HTTP:

```

<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
  <xapi:request>
    <xapi:delivery>
      <xapi:filesystem output="d:\temp\out4.pdf"/>
    </xapi:delivery>

    <xapi:document output-type="pdf">
      <xapi:template type="pdf"
        location="http://your.server:9999/templates/template1.pdf">
        <xapi:data>
          <field1>The first page data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The second page data</field1>
        </xapi:data>
      </xapi:template>
      <xapi:template type="pdf"
        location="http://your.server:9999/templates/template2.pdf">
        <xapi:data>
          <field1>The third page data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The fourth page data</field1>
        </xapi:data>
      </xapi:template>
    </xapi:document>
  </xapi:request>
</xapi:requestset>

```

### Retrieving data over HTTP

This sample builds on the previous example and shows one template with two data sources, all retrieved via HTTP; and a second template retrieved via HTTP with its two data sources embedded in the XML:

### Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
  <xapi:request>
    <xapi:delivery>
      <xapi:filesystem output="d:\temp\out5.pdf"/>
    </xapi:delivery>

    <xapi:document output-type="pdf">
      <xapi:template type="pdf"
        location="http://your.server:9999/templates/template1.pdf">
        <xapi:data location="http://your.server:9999/data/data_1.xml"/>
        <xapi:data location="http://your.server:9999/data/data_2.xml"/>
      </xapi:template>

      <xapi:template type="pdf"
        location="http://your.server:9999/templates/template2.pdf">
        <xapi:data>
          <field1>The third page data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The fourth page data</field1>
        </xapi:data>
      </xapi:template>
    </xapi:document>
  </xapi:request>
</xapi:requestset>
```

### Generating more than one output

The following sample shows the generation of two outputs: out\_1.pdf and out\_2.pdf. Note that a request element is defined for each output.

### Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
  <xapi:request>
    <xapi:delivery>
      <xapi:filesystem output="d:\temp\out_1.pdf"/>
    </xapi:delivery>
    <xapi:document output-type="pdf">
      <xapi:template type="pdf"
        location="d:\mywork\template1.pdf">
        <xapi:data>
          <field1>The first set of data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The second set of data</field1>
        </xapi:data>
      </xapi:template>
    </xapi:document>
  </xapi:request>

  <xapi:request>
    <xapi:delivery>
      <xapi:filesystem output="d:\temp\out_2.pdf"/>
    </xapi:delivery>
    <xapi:document output-type="pdf">
      <xapi:template type="pdf"
        location="d:\mywork\template2.pdf">
        <xapi:data>
          <field1>The third set of data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The fourth set of data</field1>
        </xapi:data>
      </xapi:template>
    </xapi:document>
  </xapi:request>
</xapi:requestset>
```

### Defining page numbers

The following sample shows the use of the `pagenumber` element to define page numbers on a PDF output document. The first document that is generated will begin with an initial page number value of 1. The second output document will begin with an initial page number value of 3. The `pagenumber` element can reside anywhere within the document element tags.

Note that page numbering that is applied using the `pagenumber` element will not replace page numbers that are defined in the template.



```

<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
  <xapi:request>
    <xapi:delivery>
      <xapi:filesystem output="d:\temp\out7-1.pdf"/>
    </xapi:delivery>
    <xapi:document output-type="pdf">
      <xapi:pagenumber initial-value="1" initial-page-index="1"
        x-pos="300" y-pos="20" />
      <xapi:template type="pdf"
        location="d:\mywork\template1.pdf">
        <xapi:data>
          <field1>The first page data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The second page data</field1>
        </xapi:data>
      </xapi:template>
    </xapi:document>
  </xapi:request>

  <xapi:request>
    <xapi:delivery>
      <xapi:filesystem output="d:\temp\out7-2.pdf"/>
    </xapi:delivery>
    <xapi:document output-type="pdf">
      <xapi:template type="pdf"
        location="d:\mywork\template2.pdf">
        <xapi:data>
          <field1>The third page data</field1>
        </xapi:data>
        <xapi:data>
          <field1>The fourth page data</field1>
        </xapi:data>
      </xapi:template>
      <xapi:pagenumber initial-value="3" initial-page-index="1"
        x-pos="300" y-pos="20" />
    </xapi:document>
  </xapi:request>
</xapi:requestset>

```

## Invoke Processors

The following code samples show how to invoke the document processor engine using an input file name and an input stream.

### Invoke Processors with Input File Name

Input:

- Data file name (String)
- Directory for Temporary Files (String)

**Example**

```
import oracle.apps.xdo.batch.DocumentProcessor;
.
.
.
public static void main(String[] args)
{
.
.
.
    try
    {
        // dataFile --- File path of the Document Processor XML
        // tempDir --- Temporary Directory path
        DocumentProcessor docProcessor = new DocumentProcessor(dataFile,
tempDir);
        docProcessor.process();
    }
    catch(Exception e)
    {
        e.printStackTrace();
        System.exit(1);
    }
    System.exit(0);
}
```

**Invoke Processors with InputStream**

Input:

- Data file (InputStream)
- Directory for Temporary Files (String)

### Example

```
import oracle.apps.xdo.batch.DocumentProcessor;
import java.io.InputStream;
.
.
.
public static void main(String[] args)
{
.
.
.
    try
    {
        // dataFile --- File path of the Document Processor XML
        // tempDir --- Temporary Directory path
        FileInputStream fIs = new FileInputStream(dataFile);

        DocumentProcessor docProcessor = new DocumentProcessor(fIs,
tempDir);
        docProcessor.process();
        fIs.close();
    }
    catch(Exception e)
    {
        e.printStackTrace();
        System.exit(1);
    }
    System.exit(0);
}
```

## Bursting Engine

XML Publisher's bursting engine accepts a data stream and splits it based on multiple criteria, generates output based on a template, then delivers the individual documents through the delivery channel of choice. The engine provides a flexible range of possibilities for document generation and delivery. Example implementations include:

- Invoice generation and delivery based on customer-specific layouts and delivery preference
- Financial reporting to generate a master report of all cost centers, bursting out individual cost center reports to the appropriate manager
- Generation of payslips to all employees based on one extract and delivered via e-mail

## Usage

The bursting engine is an extension of the Document Processor Engine, page 8-30 and has its own method called to invoke it. The Document Processor XML structure has been extended to handle the new components required by the bursting engine. It

supports all of the delivery functionality that the Document Processor supports using the same format. It accepts the XML data to be burst and a control file that takes the Document Processor XML format (see Hierarchy and Elements of the Document Processor XML File, page 8-30).

## Control File

The control file takes the same format as the Document Processor XML, page 8-30 with a few extensions:

- Use the attribute `select` under the `request` element to specify the element in the XML data that you wish to burst on.

### Example

```
<xapi:request select="/EMPLOYEES/EMPLOYEE">
```

- Use the attribute `id` under the lowest level of the delivery structure (for example, for the delivery element `email`, the `id` attribute belongs to the `message` element. This assigns an ID to the delivery method to be referenced later in the XML file.

### Example

```
<xapi:message id="123" to="jo.smith@company.com"
```

- Use the `delivery` attribute under the `document` element. This assigns the delivery method for the generated document as defined in the `id` attribute for the delivery element. You can specify multiple delivery channels separated by a comma.

### Example

```
<xapi:document output-type="pdf" delivery="123">
```

- Use the `filter` attribute on the `template` element. Use this to apply a layout template based on a filter on your XML data.

### Example

```
<xapi:template type="rtf" location="/usr/tmp/empGeneric.rtf">
<xapi:template type="rtf" location="usr\tmp\empDet.rtf"
filter="//EMPLOYEE[ENAME='SMITH']"/>
```

This will apply the `empDet` template only to those employees with the name "SMITH". All other employees will have the `empGeneric` template applied. This filter can use any XPATH expression to determine the rules for the template application.

## Dynamic Delivery Destination

You can reference elements in the data to derive certain delivery attributes, such as an e-mail address or fax number. Enter the value for the attribute using the following form:

`${ELEMENT}`

where `ELEMENT` is the element name from the XML data that holds the value for the attribute.

For example:

```
<xapi:message id="123" to="{EMAIL}"/>
```

At runtime the value of the `to` attribute will be set to the value of the `EMAIL` element from the input XML file.

You can also set the value of an attribute by passing a parameter to API in a Properties object.

## Dynamic Delivery Content

You can reference information in the XML data to be put into the delivery content. This takes the same format described above (that is, `{ELEMENT}`).

For example, suppose you wanted to burst a document to employees via e-mail and personalize the e-mail by using the employee's name in the subject line. Assuming the employee's name is held in an element called `ENAME`, you could use `{ENAME}` to reference the employee's name in the control file as follows:

```
subject="Employee Details for {ENAME}"
```

### Sample Control File

The following sample control file shows an example control file to split data based on an `EMPLOYEE` element and send an e-mail to each employee with their own data. The sample file is annotated.

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
<xapi:request select="/EMPLOYEES/EMPLOYEE"><!-- This sets the bursting
element i.e., EMPLOYEE -->
  <xapi:delivery>
    <xapi:email server="rgmamersmtp.oraclecorp.com" port="25"
from="xmlpadmin1@oracle.com" reply-to="reply@oracle.com">
      <xapi:message id="123" to="{EMAIL}" cc="{EMAIL_ALL}"
attachment="true" subject="Employee Details
for {ENAME}"> Mr. {ENAME}, Please review the
attached document</xapi:message><!-- This assigns a delivery id
of '123'. It also sets the e-mail
address of the employee and a cc copy to a parameter value
EMAIL_ALL; this might be a manager's e-mail. The employee's
name (ENAME) can also be used in the subject/body
of the email. --></xapi:email>
    </xapi:delivery>
    <xapi:document output-type="pdf" delivery="123">
      <xapi:template type="rtf" location="/usr/tmp/empGeneric.rtf">
        <xapi:template type="rtf" location="/usr/tmp/empDet.rtf"
filter="//EMPLOYEE[ENAME='SMITH']"><!-- Employees with the name
SMITH will have
the empDet template applied -->
      </xapi:template>
    </xapi:document>
  </xapi:request>
</xapi:requestset>
```

## Multiple Bursting Options

The bursting engine can support multiple bursting criteria and delivery options. Assume you have a report that generates data for all employees with their manager's information. You can construct a control file that will:

- Burst the employee data to each employee
- Burst a report to each manager that contains the data about his employees

You can provide a different template for each bursting level. You can therefore generate the employee report based on one template and the summary manager's report based on a different template, but still use the same data set.

To achieve this multibursting result, you must add a second `request` element to the control file structure.

### Multibursting Example

The following sample shows how to construct a control file that will burst on the EMPLOYEE level and the MANAGER level:

```

?xml version="1.0" encoding="UTF-8" ?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi"><!--
First request to burst on employee -->
  <xapi:request select="/EMPLOYEES/EMPLOYEE">
    <xapi:delivery>
      <xapi:email <<server details removed>> />
        <xapi:message id="123" <<message details removed>>
          </xapi:message>
        </xapi:email>
      <xapi:fax server="ipp://mycupsserver:631/printers/fax2">
        <xapi:number id="FAX1">916505069560</xapi:number>
        </xapi:fax>
        <xapi:print id="printer1"
          printer="ipp://mycupsserver:631/printers/printer1"
          copies="2" />
        </xapi:delivery>
      <xapi:document output-type="pdf" delivery="123">
        <xapi:template type="rtf" location="usr\tmp\empDet.rtf" />
        </xapi:document>
      </xapi:request><!--Second request to burst on department -->
    <xapi:request select="/DATA/DEPT/MANAGER">
      <xapi:delivery>
        <xapi:email server="gsmtp.oraclecorp.com" port=""
          from="XDOburstingTest@oracle.com" reply-to="reply@oracle.com">
          <xapi:message id="123" to="{MANAGER_EMAIL}"
            cc="{MANAGER_EMAIL}" attachment="true"
            subject="Department Summary for ${DEPTNO}">Please review
            the attached Department Summary for
            department ${DEPTNO}</xapi:message>
          </xapi:email>
        </xapi:delivery>
      <xapi:document output-type="rtf" delivery="123">
        <xapi:template type="rtf"
          location="d:\burst_test\deptSummary.rtf" />
        </xapi:document>
      </xapi:request>
    </xapi:requestset>
  </xapi:requestset>

```

## Bursting Listeners

The bursting engine provides a listening interface that allows you to listen to the various stages of the bursting process. Following are the supported modes that you can subscribe to:

- `beforeProcess()` - before the bursting process starts.
- `afterProcess()` - after the bursting process completes.
- `beforeProcessRequest(int requestIndex)` - before the bursting request starts. This interface provides an assigned request ID for the current request.
- `afterProcessRequest(int requestIndex)` - after the bursting request has completed; provides the request ID for the current request.
- `beforeProcessDocument(int requestIndex, int documentIndex, String deliveryId)` - before the document generation starts;

provides the request ID and a document ID.

- `afterProcessDocument(int requestIndex, int documentIndex, Vector documentOutputs)` - after the document generation completes; provides the request ID and document ID, plus a Vector list of the document objects generated in the request.
- `beforeDocumentDelivery(int requestIndex, int documentIndex, String deliveryId)` - before the documents in the request are delivered; provides the request ID, the document ID, and a delivery ID.
- `afterDocumentDelivery(int requestIndex, int documentIndex, String deliveryId, Object deliveryObject, Vector attachments)` - after the document delivery completes; provides a request ID, document ID, and delivery ID, plus a Vector list of the documents delivered in the request.

You can subscribe to any of these interfaces in your calling Java class. The listeners are useful to determine if the processing of individual documents is proceeding successfully or to start another process based on the successful completion of a request.

## Calling the Bursting API

To call the bursting API, instantiate an instance of `DocumentProcessor` class using one of the following formats:

```
DocumentProcessor(xmlCtrlInput, xmlDataInput, tmpDir)
```

where

`xmlCtrlInput` - is the control file for the bursting process. This can be a string reference to a file, an `InputStream` object, or a `Reader` object.

`xmlDataInput` - is the XML data to be burst. This can be a string reference to a file, an `InputStream` object, or a `Reader` object.

`tmpDir` - is a temporary working directory. This takes the format of a `String` object. This is optional as long as the main XML Publisher temporary directory has been set.

### Simple Example Java Class

The following is a sample Java class:



```

public class BurstingTest
{
    public BurstingTest()
    {
        try
        {
            DocumentProcessor dp = new DocumentProcessor
            ("\\burst\\burstCtrl.xml", "\\burst\\empData.xml", "\\burst");
            dp.process();
        }
        catch (Exception e)
        { System.out.println(e);
        }

        public static void main(String[] args)
        {
            BurstingTest burst1 = new BurstingTest();
        }
    }
}

```

### **Example Java Class with Listeners**

To take advantage of the bursting listeners, add the interface to the class declaration and use the `registerListener` method. Then code for the listeners you want to subscribe to as follows:

```

public class BurstingTest implements BurstingListener
{
    public BurstingTest()
    {
        try
        {
            DocumentProcessor dp = new DocumentProcessor
            ("\\burst\\burstCtrl.xml", "\\burst\\empData.xml", "\\burst");
            dp.registerListener(this);
            dp.process();
        }
        catch (Exception e)
        { System.out.println(e);
        }

        public static void main(String[] args)
        {
            BurstingTest burst1 = new BurstingTest();
        }

        public void beforeProcess() {
            System.out.println("Start of Bursting Process");
        }
        public void afterProcess()
        {
            System.out.println("End of Bursting Process");
        }

        public void beforeProcessRequest(int requestIndex)
        {
            System.out.println("Start of Process Request ID"+requestIndex);
        }
        public void afterProcessRequest(int requestIndex)
        {
            System.out.println("End of Process Request ID"+requestIndex ");
        }

        public void beforeProcessDocument(int requestIndex,int
            documentIndex)
        {
            System.out.println("Start of Process Document");
            System.out.println("    Request Index "+requestIndex);
            System.out.println("    Document Index " +documentIndex);
        }

        public void afterProcessDocument(int requestIndex,int
            documentIndex,
            Vector documentOutputs)
        {
            System.out.println("    =====End of Process Document");
            System.out.println("    Outputs :"+documentOutputs);
        }

        public void beforeDocumentDelivery(int requestIndex,int
            documentIndex,
            String deliveryId)
        {
            System.out.println("    =====Start of  Delivery");
            System.out.println("    Request Index "+requestIndex);
            System.out.println("    Document Index " +documentIndex);
            System.out.println("    DeliveryId " +deliveryId);
        }
    }
}

```

```

public void afterDocumentDelivery(int requestIndex,int documentIndex,
    String deliveryId,Object deliveryObject,Vector attachments)
{
    System.out.println("    =====End of  Delivery");
    System.out.println("    Attachments : "+attachments);

}

}

```

### **Passing a Parameter**

To pass a parameter holding a value to be used in the control file for delivery, add the following code:

```

...
Properties prop= new Properties();
prop.put("user-variable:ADMIN_EMAIL","jo.smith@company.com");
dp.setConfig(prop);
dp.process();
...

```

## **Bursting Control File Examples**

All of the examples in this section use the following XML data source:

```

<?xml version="1.0" encoding="UTF-8"?>
<DATA>
<DEPTS>
  <DEPT>
    <DEPTNO>20</DEPTNO>
    <NAME>Accounting</NAME>
    <MANAGER_EMAIL>tdexter@mycomp.com</MANAGER_EMAIL>
    <EMPLOYEES>
      <EMPLOYEE>
        <EMPNO>7369</EMPNO>
        <ENAME>SMITH</ENAME>
        <JOB>CLERK</JOB>
        <MGR>7902</MGR>
        <HIREDATE>1980-12-17T00:00:00.000-08:00</HIREDATE>
        <SAL>800</SAL>
        <DEPTNO>20</DEPTNO>
        <EMAIL>jsmith@mycomp.com</EMAIL>
      </EMPLOYEE>
      <EMPLOYEE>
        <EMPNO>7566</EMPNO>
        <ENAME>JONES</ENAME>
        <JOB>MANAGER</JOB>
        <MGR>7839</MGR>
        <HIREDATE>1981-04-02T00:00:00.000-08:00</HIREDATE>
        <SAL>2975</SAL>
        <DEPTNO>20</DEPTNO>
        <EMAIL>jjones@mycomp.com</EMAIL>
      </EMPLOYEE>
    </EMPLOYEES>
  </DEPT>
  <DEPT>
    <DEPTNO>30</DEPTNO>
    <NAME>Sales</NAME>
    <MANAGER_EMAIL>dsmith@mycomp.com</MANAGER_EMAIL>
    <EMPLOYEES>
      <EMPLOYEE>
        <EMPNO>7788</EMPNO>
        <ENAME>SCOTT</ENAME>
        <JOB>ANALYST</JOB>
        <MGR>7566</MGR>
        <HIREDATE>1982-12-09T00:00:00.000-08:00</HIREDATE>
        <SAL>3000</SAL>
        <DEPTNO>20</DEPTNO>
        <EMAIL>jscott@mycomp.com</EMAIL>
      </EMPLOYEE>
      <EMPLOYEE>
        <EMPNO>7876</EMPNO>
        <ENAME>ADAMS</ENAME>
        <JOB>CLERK</JOB>
        <MGR>7788</MGR>
        <HIREDATE>1983-01-12T00:00:00.000-08:00</HIREDATE>
        <SAL>1100</SAL>
        <EMAIL>jadams@mycomp.com</EMAIL>
      </EMPLOYEE>
    </EMPLOYEES>
  </DEPT>
</DEPTS>
</DATA>

```

### Example 1 - Bursting Employee Data to Employees via E-mail

The following sample shows how to apply a template (empDet.rtf) to every employee's

data, generate a PDF document, and deliver the document to each employee via e-mail.

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
<xapi:request select="/DATA/DEPTS/DEPT/EMPLOYEES/EMPLOYEE"> <!-- Burst
on employee element - >
  <xapi:delivery>
    <xapi:email server="my.smtp.server" port="25"
      from="xmlpadmin@mycomp.com" reply-to="">
      <xapi:message id="123" to="{EMAIL}"
<!-- Set the id for the delivery method - > <!-- Use the employees
EMAIL element to email the document to
the employee - > cc="{ADMIN_EMAIL}"
<!-- Use the ADMIN_EMAIL parameter to CC the document
to the administrator - > attachment="true" subject="Employee
Details for ${ENAME}">
  Mr. ${ENAME}, Please review the attached document</xapi:message><!--
Embed the employees name into the email message - >
</xapi:email>
  </xapi:delivery>
  <xapi:document output-type="pdf" delivery="123"><!--Specify the
delivery method id to be used - >
    <xapi:template type="rtf"
      location="\usr\empDet.rtf"></xapi:template>
    </xapi:document>
  </xapi:request>
</xapi:requestset>
```

## Example 2 - Bursting Employee Data to Employees via Multiple Delivery Channels and Conditionally Using Layout Templates

This sample shows how to burst, check the employee name, and generate a PDF using the appropriate template. The documents will then be e-mailed and printed.

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi" >
  <xapi:globalData location="stream">
  </xapi:globalData >
  <xapi:request select="/DATA/DEPTS/DEPT/EMPLOYEES/EMPLOYEE">
    <xapi:delivery>
      <xapi:email server="my.smtp.server" port=""
        from="xmlpserver@oracle.com"
        reply-to="reply@oracle.com">
        <xapi:message id="123" to="{EMAIL}" cc="" attachment="true"
          subject="Employee Details for ${ENAME}"> Mr. ${ENAME},
          Please review the attached document</xapi:message>
      </xapi:email>
      <xapi:print id="printer1"
        printer="ipp://ipgpcl:631/printers/printer1" copies="2" /><!-- Add an
id for this delivery method i.e. printer1 - > </xapi:delivery>
      <xapi:document output-type="pdf" delivery="printer1,123"><!--
Deliver to printer and email - > <xapi:template type="rtf"
location="/usr/empDetSmith.rtf"
      filter="//EMPLOYEE[ENAME='SMITH']"><!-- Specify template to be
used for employees called SMITH - >
      </xapi:template>
      <xapi:template type="rtf" location="/usr/empSummary.rtf"><!--
Default template to be used - >
      </xapi:template>
    </xapi:document>
  </xapi:request>
</xapi:requestset>
```

### Example 3 - Bursting Employee Data to Employees and Their Manager

This sample shows how to burst an e-mail with a PDF attachment to all employees using the empDet template. It will also burst an employee summary PDF to the manager of each department via e-mail.

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
  <xapi:request select="/DATA/DEPTS/DEPT/EMPLOYEES/EMPLOYEE">
    <xapi:delivery>
      <xapi:email server="my.smtp.server" port=""
        from="xmlpserver@oracle.com" reply-to="">
        <xapi:message id="123" to="{EMAIL}" cc="{EMAIL}"
          attachment="true"
          subject="Employee Details for ${ENAME}"> Mr. ${ENAME},
            Please review the attached document</xapi:message>
        </xapi:email>
      </xapi:delivery>
      <xapi:document output-type="pdf" delivery="123">
        <xapi:template type="rtf"
          location="/usr/empDet.rtf"></xapi:template>
      </xapi:document>
    </xapi:request>
    <xapi:request select="/DATA/DEPTS/DEPT"><!-- Second request created to
burst the same dataset to the
manager based on the DEPT element -->
      <xapi:delivery>
        <xapi:email server="my.smtp.server" port=""
          from="xmlpserver@oracle.com" reply-to="">
        <xapi:message id="456" to="{MANAGER_EMAIL}"
          cc="{MANAGER_EMAIL}" attachment="true" subject="Department
          Summary for ${DEPTNO}"> Please review the attached
          Department Summary for department ${DEPTNO}</xapi:message>
        </xapi:email>
      </xapi:delivery>
      <xapi:document output-type="rtf" delivery="456">
        <xapi:template type="rtf"
          location="\usr\deptSumm.rtf"></xapi:template>
      </xapi:document>
    </xapi:request>
  </xapi:requestset>
```

## XML Publisher Properties

The FO Processor supports PDF security and other properties that can be applied to your final documents. Security properties include making a document unprintable and applying password security to an encrypted document.

Other properties allow you to define font subsetting and embedding. If your template uses a font that would not normally be available to XML Publisher at runtime, you can use the font properties to specify the location of the font. At runtime XML Publisher will retrieve and use the font in the final document. For example, this property might be used for check printing for which a MICR font is used to generate the account and routing numbers on the checks.

See XML Publisher Properties, page 7-4 for the full list of properties.

## Setting Properties

The properties can be set in two ways:

- At runtime, specify the property as a Java Property object to pass to the FO Processor.
- Set the property in a configuration file.
- Set the property in the template (RTF templates only). See Setting Properties, page 2-91 in the RTF template for this method.

### Passing Properties to the FO Engine

To pass a property as a Property object, set the name/value pair for the property prior to calling the FO Processor, as shown in the following example:

Input:

- XML file name (String)
- XSL file name (String)

Output:

- PDF file name (String)

### Example

```
import oracle.apps.xdo.template.FOProcessor;
.
.
.
public static void main(String[] args)
{

    FOProcessor processor = new FOProcessor();
    processor.setData(args[0]);      // set XML input file
    processor.setTemplate(args[1]); // set XSL input file
    processor.setOutput(args[2]); //set (PDF) output file
    processor.setOutputFormat(FOProcessor.FORMAT_PDF);
    Properties prop = new Properties();
    /* PDF Security control: */
    prop.put("pdf-security", "true");
    /* Permissions password: */
    prop.put("pdf-permissions-password", "abc");
    /* Encryption level: */
    prop.put("pdf-encryption-level", "0");
    processor.setConfig(prop);
    // Start processing
    try
    {
        processor.generate();
    }
    catch (XDOException e)
    {
        e.printStackTrace();
        System.exit(1);
    }

    System.exit(0);
}
```

### Passing a Configuration File to the FO Processor

The following code shows an example of passing the location of a configuration file.

Input:

- XML file name (String)
- XSL file name (String)

Output:

- PDF file name (String)



```

import oracle.apps.xdo.template.FOProcessor;
.
.
.
public static void main(String[] args)
{
    FOProcessor processor = new FOProcessor();
    processor.setData(args[0]); // set XML input file
    processor.setTemplate(args[1]); // set XSL input file
    processor.setOutput(args[2]); //set (PDF) output file
    processor.setOutputFormat(FOProcessor.FORMAT_PDF);
    processor.setConfig("/tmp/xmlpconfig.xml");
    // Start processing
    try
    {
        processor.generate();
    } catch (XDOException e)
    {
        e.printStackTrace();
        System.exit(1);
    }
    System.exit(0);
}

```

### Passing Properties to the Document Processor

Input:

- Data file name (String)
- Directory for Temporary Files (String)

Output:

- PDF File

### Example

```
import oracle.apps.xdo.batch.DocumentProcessor;

.
.
.
public static void main(String[] args)
{
.
.
.
    try
    {
        // dataFile --- File path of the Document Processor XML
        // tempDir --- Temporary Directory path
        DocumentProcessor docProcessor = new DocumentProcessor(dataFile,
tempDir);
        Properties prop = new Properties();
        /* PDF Security control: */
        prop.put("pdf-security", "true");
        /* Permissions password: */
        prop.put("pdf-permissions-password", "abc");
        /* encryption level: */
        prop.put("pdf-encryption-level", "0");
        processor.setConfig(prop);
        docProcessor.process();
    }
    catch(Exception e)
    {
        e.printStackTrace();
        System.exit(1);
    }
    System.exit(0);
}
```

## Advanced Barcode Font Formatting Implementation

For the advanced formatting to work in the template, you must provide a Java class with the appropriate methods to format the data at runtime. Many font vendors offer the code with their fonts to carry out the formatting; these must be incorporated as methods into a class that is available to the XML Publisher formatting libraries at runtime. There are some specific interfaces that you must provide in the class for the library to call the correct method for encoding.

**Note:** See Advanced Barcode Formatting, page 2-118 for the setup required in the RTF template.

You must implement the following three methods in this class:

```

/**
 * Return a unique ID for this barcode encoder
 * @return the id as a string
 */
public String getVendorID();

/**
 * Return true if this encoder support a specific type of barcode
 * @param type the type of the barcode
 * @return true if supported
 */
public boolean isSupported(String type);

/**
 * Encode a barcode string by given a specific type
 * @param data the original data for the barcode
 * @param type the type of the barcode
 * @return the formatted data
 */
public String encode(String data, String type);

```

Place this class in the classpath for the middle tier JVM in which XML Publisher is running.

**Note:** For E-Business Suite users, the class must be placed in the classpath for the middle tier and any concurrent nodes that are present.

If in the register-barcode-vendor command the `barcode_vendor_id` is not provided, XML Publisher will call the `getVendorID()` and use the result of the method as the ID for the vendor.

The following is an example class that supports the code128 a, b and c encodings:

**Important:** The following code sample can be copied and pasted for use in your system. Note that due to publishing constraints you will need to correct line breaks and ensure that you delete quotes that display as "smart quotes" and replace them with simple quotes.

## Example

```
package oracle.apps.xdo.template.rtf.util.barcode;

import java.util.Hashtable;
import java.lang.reflect.Method;
import oracle.apps.xdo.template.rtf.util.XDOBarcodeEncoder;
import oracle.apps.xdo.common.log.Logger;
// This class name will be used in the register vendor
// field in the template.

public class BarcodeUtil implements XDOBarcodeEncoder
// The class implements the XDOBarcodeEncoder interface
{
    // This is the barcode vendor id that is used in the
    // register vendor field and format-barcode fields
    public static final String BARCODE_VENDOR_ID = "XMLPBarVendor";
    // The hashtable is used to store references to
    // the encoding methods
    public static final Hashtable ENCODERS = new Hashtable(10);
    // The BarcodeUtil class needs to be instantiated
    public static final BarcodeUtil mUtility = new BarcodeUtil();
    // This is the main code that is executed in the class,
    // it is loading the methods for the encoding into the hashtable.
    // In this case we are loading the three code128 encoding
    // methods we have created.
    static {
        try {
            Class[] clazz = new Class[] { ""getClass() };

            ENCODERS.put("code128a",mUtility.getClass().getMethod("code128a",
clazz));
            ENCODERS.put("code128b",mUtility.getClass().getMethod("code128b",
clazz));
            ENCODERS.put("code128c",mUtility.getClass().getMethod("code128c",
clazz));
        } catch (Exception e) {
            // This is using the XML Publisher logging class to push
            // errors to the XMLP log file.
            Logger.log(e,5);
        }
    }
}
```

```

// The getVendorID method is called from the template layer
// at runtime to ensure the correct encoding method are used
    public final String getVendorID()
    {
        return BARCODE_VENDOR_ID;
    }
//The isSupported method is called to ensure that the
// encoding method called from the template is actually
// present in this class.
// If not then XMLP will report this in the log.
    public final boolean isSupported(String s)
    {
        if(s != null)
            return ENCODERS.containsKey(s.trim().toLowerCase());
        else
            return false;
    }

// The encode method is called to then call the appropriate
// encoding method, in this example the code128a/b/c methods.

    public final String encode(String s, String s1)
    {
        if(s != null && s1 != null)
        {
            try
            {
                Method method =
(Method)ENCODERS.get(s1.trim().toLowerCase());
                if(method != null)
                    return (String)method.invoke(this, new Object[] {
                        s
                    });
                else
                    return s;
            }
            catch(Exception exception)
            {
                Logger.log(exception,5);
            }
            return s;
        } else
        {
            return s;
        }
    }

/** This is the complete method for Code128a */

    public static final String code128a( String DataToEncode )
    {
        char C128_Start = (char)203;
        char C128_Stop = (char)206;
        String Printable_string = "";
        char CurrentChar;
        int CurrentValue=0;
        int weightedTotal=0;
        int CheckDigitValue=0;
        char C128_CheckDigit='w';

        DataToEncode = DataToEncode.trim();

```

```

weightedTotal = ((int)C128_Start) - 100;
for( int i = 1; i <= DataToEncode.length(); i++ )
{
    //get the value of each character
    CurrentChar = DataToEncode.charAt(i-1);
    if( ((int)CurrentChar) < 135 )
        CurrentValue = ((int)CurrentChar) - 32;
    if( ((int)CurrentChar) > 134 )
        CurrentValue = ((int)CurrentChar) - 100;

    CurrentValue = CurrentValue * i;
    weightedTotal = weightedTotal + CurrentValue;
}
//divide the WeightedTotal by 103 and get the remainder, //this is
the CheckDigitValue
CheckDigitValue = weightedTotal % 103;
if( (CheckDigitValue < 95) && (CheckDigitValue > 0) )
    C128_CheckDigit = (char)(CheckDigitValue + 32);
if( CheckDigitValue > 94 )
    C128_CheckDigit = (char)(CheckDigitValue + 100);
if( CheckDigitValue == 0 ){
    C128_CheckDigit = (char)194;
}

Printable_string = C128_Start + DataToEncode + C128_CheckDigit +
C128_Stop + " ";
return Printable_string;
}

```

```

/** This is the complete method for Code128b ***/

public static final String code128b( String DataToEncode )
{
    char C128_Start = (char)204;
    char C128_Stop = (char)206;
    String Printable_string = "";
    char CurrentChar;
    int CurrentValue=0;
    int weightedTotal=0;
    int CheckDigitValue=0;
    char C128_CheckDigit='w';

    DataToEncode = DataToEncode.trim();
    weightedTotal = ((int)C128_Start) - 100;
    for( int i = 1; i <= DataToEncode.length(); i++ )
    {
//get the value of each character
        CurrentChar = DataToEncode.charAt(i-1);
        if( ((int)CurrentChar) < 135 )
            CurrentValue = ((int)CurrentChar) - 32;
        if( ((int)CurrentChar) > 134 )
            CurrentValue = ((int)CurrentChar) - 100;

        CurrentValue = CurrentValue * i;
        weightedTotal = weightedTotal + CurrentValue;
    }
//divide the WeightedTotal by 103 and get the remainder, //this is
the CheckDigitValue
    CheckDigitValue = weightedTotal % 103;
    if( (CheckDigitValue < 95) && (CheckDigitValue > 0) )
        C128_CheckDigit = (char)(CheckDigitValue + 32);
    if( CheckDigitValue > 94 )
        C128_CheckDigit = (char)(CheckDigitValue + 100);
    if( CheckDigitValue == 0 ){
        C128_CheckDigit = (char)194;
    }

    Printable_string = C128_Start + DataToEncode + C128_CheckDigit +
    C128_Stop + " ";
    return Printable_string;
}

/** This is the complete method for Code128c **/

public static final String code128c( String s )
{
    char C128_Start = (char)205;
    char C128_Stop = (char)206;
    String Printable_string = "";
    String DataToPrint = "";
    String OnlyCorrectData = "";
    int i=1;
    int CurrentChar=0;
    int CurrentValue=0;
    int weightedTotal=0;
    int CheckDigitValue=0;
    char C128_CheckDigit='w';
    DataToPrint = "";
    s = s.trim();

```

```

for(i = 1; i <= s.length(); i++ )
{
    //Add only numbers to OnlyCorrectData string
    CurrentChar = (int)s.charAt(i-1);
    if((CurrentChar < 58) && (CurrentChar > 47))
    {
        OnlyCorrectData = OnlyCorrectData + (char)s.charAt(i-1);
    }
    s = OnlyCorrectData;
    //Check for an even number of digits, add 0 if not even
    if( (s.length() % 2) == 1 )
    {
        s = "0" + s;
    }
    //<<<< Calculate Modulo 103 Check Digit and generate
    // DataToPrint >>>>Set WeightedTotal to the Code 128 value of
// the start character
    weightedTotal = ((int)C128_Start) - 100;
    int WeightValue = 1;
    for( i = 1; i <= s.length(); i += 2 )
    {
        //Get the value of each number pair (ex: 5 and 6 = 5*10+6 =56) //And
assign the ASCII values to DataToPrint
        CurrentChar = (((int)s.charAt(i-1))-48)*10 + (((int)s.charAt(i))-48);
        if((CurrentChar < 95) && (CurrentChar > 0))
            DataToPrint = DataToPrint + (char)(CurrentChar + 32);
        if( CurrentChar > 94 )
            DataToPrint = DataToPrint + (char)(CurrentChar + 100);
        if( CurrentChar == 0)
            DataToPrint = DataToPrint + (char)194;
        //multiply by the weighting character
        //add the values together to get the weighted total
        weightedTotal = weightedTotal + (CurrentChar * WeightValue);
        WeightValue = WeightValue + 1;
    }
    //divide the WeightedTotal by 103 and get the remainder, //this is
the CheckDigitValue
    CheckDigitValue = weightedTotal % 103;
    if((CheckDigitValue < 95) && (CheckDigitValue > 0))
        C128_CheckDigit = (char)(CheckDigitValue + 32);
    if( CheckDigitValue > 94 )
        C128_CheckDigit = (char)(CheckDigitValue + 100);
    if( CheckDigitValue == 0 ){
        C128_CheckDigit = (char)194;
    }
    Printable_string = C128_Start + DataToPrint + C128_CheckDigit +
    C128_Stop + " ";
    Logger.log(Printable_string,5);
    return Printable_string;
}
}

```

Once you create the class and place it in the correct classpath, your template creators can start using it to format the data for barcodes. You must give them the following information to include in the template commands:

- The class name and path.

In this example:



```
oracle.apps.xdo.template.rtf.util.barcoder.BarcodeUtil
```

- The barcode vendor ID you created.

In this example: XMLPBarVendor

- The available encoding methods.

In this example, code128a, code128b and code128c They can then use this information to successfully encode their data for barcode output.

They can then use this information to successfully encode their data for barcode output.



---

## Supported XSL-FO Elements

### Supported XSL-FO Elements

The following table lists the XSL-FO elements supported in this release of BI Publisher. For each element the supported content elements and attributes are listed. If elements have shared supported attributes, these are noted as a group and are listed in the subsequent table, Property Groups. For example, several elements share the content element `inline`. Rather than list the `inline` properties each time, each entry notes that "inline-properties" are supported. The list of inline-properties can then be found in the Property Groups table.

Element	Supported Content Elements	Supported Attributes
basic-link	external-graphic	inline-properties
	inline	external-destination
	leader	internal-destination
	page-number	
	page-number-citation	
	basic-link	
	block	
	block-container	
	table	
	list-block	
	wrapper	

Element	Supported Content Elements	Supported Attributes
bidi-override	bidi-override external-graphic instream-foreign-object inline leader page-number page-number-citation basic-link	inline-properties
block	external-graphic inline page-number page-number-citation basic-link block block-container table list-block wrapper	block-properties
block-container	block block-container table list-block wrapper	block-properties
bookmark-tree	bookmark	N/A

Element	Supported Content Elements	Supported Attributes
bookmark	bookmark	external-destination
	bookmark-title	internal-destination starting-state
bookmark-title	N/A	color font-style font-weight
conditional-page-master-reference	N/A	master-reference page-position <ul style="list-style-type: none"> <li>first</li> <li>last</li> <li>rest</li> <li>any</li> <li>inherit</li> </ul> odd-or-even <ul style="list-style-type: none"> <li>odd</li> <li>even</li> <li>any</li> <li>inherit</li> </ul> blank-or-not-blank <ul style="list-style-type: none"> <li>blank</li> <li>not-blank</li> <li>any</li> <li>inherit</li> </ul>

Element	Supported Content Elements	Supported Attributes
external-graphic	N/A	graphic-properties src
flow	block block-container table list-block wrapper	flow-properties
inline	external-graphic inline leader page-number page-number-citation basic-link block block-container table wrapper	inline-properties
instream-foreign-object	N/A	graphic-properties
layout-master-set	page-sequence-master simple-page-master simple-page-master page-sequence-master	N/A
leader	N/A	inline-properties
list-block	list-item	block-properties

Element	Supported Content Elements	Supported Attributes
list-item	list-item-label	block-properties
	list-item-body	
list-item-body	block	block-properties
	block-container	
	table	
	list-block	
	wrapper	
list-item-label	block	block-properties
	block-container	
	table	
	list-block	
	wrapper	
page-number	N/A	empty-inline-properties
page-number-citation	N/A	empty-inline-properties
		ref-id

Element	Supported Content Elements	Supported Attributes
page-sequence	static-content	inheritable-properties
	flow	id master-reference initial-page-number force-page-count <ul style="list-style-type: none"> <li>• auto</li> <li>• end-on-even</li> <li>• end-on-odd</li> <li>• end-on-even-layout</li> <li>• end-on-odd-layout</li> <li>• no-force</li> <li>• inherit</li> </ul> format
page-sequence-master	single-page-master-reference	master-name
	repeatable-page-master-reference	
	repeatable-page-master-alternatives	
region-after	N/A	side-region-properties
region-before	N/A	side-region-properties
region-body	N/A	region-properties
		margin-properties-CSS
		column-count
region-end	N/A	side-region-properties



Element	Supported Content Elements	Supported Attributes
region-start	N/A	side-region-properties
repeatable-page-master-alternatives	conditional-page-master-reference	maximum-repeats
repeatable-page-master-reference	N/A	master-reference maximum-repeats
root	bookmark-tree layout-master-set page-sequence	inheritable-properties

Element	Supported Content Elements	Supported Attributes
simple-page-master	region-body	margin-properties-CSS
	region-before	master-name
	region-after	page-height
	region-start	page-width
	region-end	reference-orientation <ul style="list-style-type: none"> <li>• 0</li> <li>• 90</li> <li>• 180</li> <li>• 270</li> <li>• -90</li> <li>• -180</li> <li>• -270</li> <li>• 0deg</li> <li>• 90deg</li> <li>• 180deg</li> <li>• 270deg</li> <li>• -90deg</li> <li>• -180deg</li> <li>• -270deg</li> <li>• inherit</li> </ul> writing-mode <ul style="list-style-type: none"> <li>• lr-tb</li> </ul>

Element	Supported Content Elements	Supported Attributes
single-page-master-reference	N/A	master-reference
static-content	block block-container table wrapper	flow-properties
table	table-column table-header table-footer table-body	block-properties
table-body	table-row	inheritable-properties id
table-cell	block block-container table list-block wrapper	block-properties number-columns-spanned number-rows-spanned
table-column	N/A	inheritable-properties column-number column-width number-columns-repeated
table-footer	table-row	inheritable-properties id
table-header	table-row	inheritable-properties id

Element	Supported Content Elements	Supported Attributes
table-row	table-cell	inheritable-properties id
wrapper	inline page-number page-number-citation basic-link block block-container table wrapper	inheritable-properties id

### Property Groups Table

The following table lists the supported properties belonging to the attribute groups defined in the preceding table.

Property Group	Properties
area-properties	overflow (visible, hidden)
	reference-orientation
	• 0
	• 90
	• 180
	• 270
	• -90
	• -180
	• -270
	• 0deg
	• 90deg
	• 180deg
	• 270deg
	• -90deg
	• -180deg
	• -270deg
	• inherit
block-properties	writing-mode (lr-tb, rl-tb, lr, rl)
	baseline-shift (baseline, sub, super)
	vertical-align
	inheritable-properties
	id

Property Group	Properties
border-padding-background-properties	background-color background-image background-position-vertical background-position-horizontal border border-after-color border-after-style (none, dotted, dashed, solid, double) border-after-width border-before-color border-before-style (none, solid) border-before-width border-bottom border-bottom-color border-bottom-style (none, dotted, dashed, solid, double) border-bottom-width border-color border-end-color border-end-style (none, dotted, dashed, solid, double) border-end-width border-left border-left-color border-left-style (none, dotted, dashed, solid, double) border-left-width border-right border-right-color border-right-style (none, dotted, dashed, solid, double) border-right-width border-start-color

Property Group	Properties
	border-start-style (none, dotted, dashed, solid, double)
	border-start-width
	border-top
	border-top-color
	border-top-style (none, dotted, dashed, solid, double)
	border-top-width
	border-width
	padding
	padding-after
	padding-before
	padding-bottom
	padding-end
	padding-left
	padding-right
	padding-start
	padding-top
box-size-properties	height
	width
character-properties	font-properties
	text-decoration
empty-inline-properties	character-properties
	border-padding-background-properties
	id
	color

Property Group	Properties
flow-properties	inheritable-properties
	id
	flow-name
font-properties	font-family
	font-size
	font-style (normal, italic, oblique)
	font-weight (normal, bold)
	table-omit-header-at-break (TRUE, FALSE, inherit)
	table-omit-footer-at-break (TRUE, FALSE, inherit)
graphic-properties	border-padding-background-properties
	margin-properties-inline
	box-size-properties
	font-properties
	keeps-and-breaks-properties-atomic
	id



Property Group	Properties
inheritable-properties	border-padding-background-properties box-size-properties margin-properties-inline area-properties character-properties line-related-properties leader-properties keeps-and-breaks-properties-block color absolute-position <ul style="list-style-type: none"> <li>• auto</li> <li>• absolute</li> <li>• fixed</li> <li>• inherit</li> </ul>
inline-properties	inheritable-properties id
keeps-and-breaks-properties-atomic	break-after (auto, column, page) break-before (auto,column) keep-with-next keep-with-next.within-page
keeps-and-breaks-properties-block	keeps-and-breaks-properties-inline

Property Group	Properties
keeps-and-breaks-properties-inline	keeps-and-breaks-properties-atomic keep-together keep-together.within-line keep-together.within-column keep-together.within-page
leader-properties	leader-pattern (rule, dots) leader-length leader-length.optimum (dotted, dashed, solid, double) rule-thickness
line-related-properties	text-align (start, center, end, justify, left, right, inherit) text-align-last (start, center, end, justify, left, right, inherit) text-indent linefeed-treatment (ignore, preserve, treat-as-space, treat-as-zero-width-space, inherit ) white-space-treatment (ignore, preserve, ignore-if-before-linefeed, ignore-if-after-linefeed, ignore-if-surrounding-linefeed, inherit) white-space-collapse (FALSE, TRUE, inherit) wrap-option (no-wrap, wrap, inherit) direction (ltr)
margin-properties-block	margin-properties-CSS space-after space-after.optimum space-before space-before.optimum start-indent end-indent

Property Group	Properties
margin-properties-CSS	margin margin-bottom margin-left margin-right margin-top
margin-properties-inline	margin-properties-block space-start space-start.optimum space-end space-end.optimum position <ul style="list-style-type: none"> <li>static</li> <li>relative</li> <li>absolute</li> <li>fixed</li> <li>inherit</li> </ul> top left
region-properties	border-padding-background-properties area-properties region-name
side-region-properties	region-properties extent



---

# Index

## A

---

alignment  
    RTF template, 2-42

## B

---

background support  
    RTF templates, 2-47  
barcode formatting, 2-118  
    APIs, 8-58  
bidirectional language alignment  
    RTF template, 2-42  
body tags  
    PDF template, 3-8  
    RTF template, 2-16  
bookmarks  
    generating PDF bookmarks from an RTF  
    template, 2-57  
    inserting in RTF templates, 2-54  
brought forward/carried forward page totals, 2-  
75  
bursting engine, 8-43

## C

---

calculations in PDF template, 3-13  
calendar profile option, 2-116  
calendar specification, 2-116  
cell highlighting  
    conditional in RTF templates, 2-71  
charts  
    building in RTF templates, 2-19  
check box placeholder

    creating in PDF template, 3-5  
check box support  
    RTF templates, 2-58  
choose statements, 2-65  
clip art support, 2-30  
columns  
    fixed width in tables, 2-43  
conditional columns  
    rtf template, 2-66  
conditional formatting, 2-62  
    table rows, 2-69  
conditional formatting features, 2-62  
configuration file  
    <properties> element, 7-3  
    <root> element, 7-3  
    structure, 7-3  
context command, 2-123  
cross-tab reports, 2-95

## D

---

date fields in RTF templates, 2-45  
drawing support, 2-30  
drop-down form field support  
    RTF templates, 2-59  
dynamic data columns, 2-98  
    example, 2-99  
dynamic table of contents in RTF template, 2-57

## E

---

end on even page, 2-54  
etext data tables, 4-6  
etext template command rows, 4-6

etext template setup command table, 4-16  
even page  
    force report to end on, 2-54

---

## F

fixed-width columns  
    RTF templates, 2-43  
FO  
    supported elements, A-1  
FO elements  
    using in RTF templates, 2-128, 5-7  
font definitions  
    configuration file, 7-14  
font fallback mechanism, 7-17  
fonts  
    external, 2-117  
    setting up, 2-117  
footers  
    RTF template, 2-15  
for-each-group XSL 2.0 standard, 2-82  
formatting options in PDF templates, 3-5  
form field method  
    inserting placeholders, 2-9  
form field properties options in PDF template, 3-5  
form fields in the PDF template, 3-3

---

## G

groups  
    basic RTF method, 2-13  
    defining in PDF template, 3-7  
    defining in RTF template, 2-12  
        syntax, 2-12  
    defining in RTF templates, 2-6  
    form field method, 2-13  
    grouping scenarios in RTF template, 2-12  
    in RTF templates, 2-5

---

## H

headers and footers  
    different first page , 2-16  
    different odd and even pages, 2-16  
    inserting placeholders, 2-15  
    multiple, 2-16  
    resetting within one output file, 2-93

    RTF template, 2-15  
hidden text  
    support in RTF templates, 2-42  
horizontal table break, 2-99  
hyperlinks  
    bookmarks, 2-54  
    dynamic, 2-54  
    inserting in RTF template, 2-54  
    internal, 2-54  
    static, 2-54

---

## I

if statements, 2-63, 2-63  
IF statements  
    in free-form text, 2-63  
if-then-else statements, 2-64  
images  
    including in RTF template, 2-17  
IN predicate  
    If-Then-Else control structure  
        e-text templates, 4-28

---

## L

last page  
    support for special content, 2-51  
locales  
    configuration file, 7-16

---

## M

markup  
    adding to the PDF template, 3-3  
    adding to the RTF template, 2-7  
merging PDF files, 8-27  
multicolumn page support, 2-45  
multiple headers and footers  
    RTF template, 2-16

---

## N

Namespace support in RTF template, 2-122  
native page breaks and page numbering, 2-41  
nulls  
    how to test for in XML data, 2-81

---

## O

Out of memory error

- avoiding, 7-5
- overflow data in PDF templates, 3-16
- overview, 1-1

## P

---

- page breaks
  - PDF templates, 3-9
  - RTF template, 2-41, 2-49
- page breaks and page numbering
  - native support, 2-41
- page number
  - setting initial
    - RTF templates, 2-50
- page numbers
  - PDF templates, 3-9
  - restarting within one output file, 2-93
  - RTF template, 2-42
- page totals
  - brought forward/carried forward, 2-75
  - inserting in RTF template, 2-73
- PDF files
  - merging, 8-27
- PDF template
  - adding markup, 3-3
  - placeholders
    - types of, 3-4
- PDF templates
  - completed example, 3-13
  - creating from downloaded file, 3-16
  - defining groups, 3-7
  - definition of, 3-1
  - overflow data, 3-16
  - page breaks, 3-9
  - page numbering, 3-9
  - placeholders
    - check box, 3-5
    - radio button group, 3-6
    - text, 3-4
  - placement of repeating fields at runtime, 3-14
  - runtime behaviors, 3-14
  - sample purchase order template, 3-2
  - saving as Adobe Acrobat 5.0 compatible, 3-1
  - sources for document templates, 3-2
  - supported modes, 3-1
  - when to use, 3-1
- placeholders

- basic RTF method, 2-8, 2-8
- form field RTF method, 2-8, 2-9
- in PDF templates, 3-3
- in RTF templates, 2-5
  - defining, 2-6, 2-8
- inserting in the header and footer of RTF template, 2-15
- PDF templates
  - check box, 3-5
  - radio button group, 3-6
  - text, 3-4
  - types of, 3-4
- predefined fonts, 7-18
- properties
  - setting at template level, 2-91
- properties element
  - configuration file, 7-3

## R

---

- radio button group
  - creating in PDF templates, 3-6
- regrouping, 2-82
- repeating elements
  - See* groups
- Rich Text Format (RTF)
  - definition, 2-1
- row breaking
  - preventing in RTF templates, 2-43
- row formatting
  - conditional, 2-69
- RTF placeholders
  - syntax, 2-8
- RTF template
  - adding markup, 2-7
  - applying design elements, 2-7
  - definition, 2-1
  - designing, 2-2
  - groups, 2-5
  - including images, 2-17
  - native formatting features, 2-41
  - placeholders, 2-5
  - prerequisites, 2-2
  - sample template design, 2-3
  - supported modes, 2-1
    - basic method, 2-1
    - form field method, 2-1

- using XSL or XSL:FO, 2-2
- RTF template design
  - headers and footers, 2-15
- RTF template placeholders, 2-8
- running totals
  - RTF templates, 2-79

## S

---

- sample RTF template
  - completed markup, 2-11
- section context command, 2-93
- setting the initial page number
  - RTF templates, 2-50
- shape support, 2-30
- sorting
  - RTF template, 2-81
- SQL functions
  - using in RTF templates, 2-119
  - XML Publisher syntax for, 5-1
- SQL functions extended for XML Publisher, 5-1
- syntax
  - RTF template placeholder, 2-8

## T

---

- table borders
  - configure overlapping borders, 7-11
- table features
  - fixed-width columns, 2-43
  - preventing rows breaking across pages
    - RTF template, 2-43
  - text truncation, 2-44
- table features
  - repeating table headers
    - RTF template, 2-43
  - RTF template, 2-42
- table of contents support
  - RTF template, 2-57
  - dynamic TOC, 2-57
- tables
  - horizontal table break, 2-99
- Template Builder, 2-2
- text placeholder
  - creating in PDF template, 3-4
- text truncation in tables, 2-44
- totals
  - brought forward/carried forward, 2-75

- inserting page totals in RTF template, 2-73
- running
  - RTF templates, 2-79
- translatable templates, 6-1

## U

---

- updateable variables
  - RTF templates, 2-88

## V

---

- variables
  - RTF templates, 2-88

## W

---

- watermarks
  - RTF templates, 2-47

## X

---

- XML data file
  - example, 2-4
- XML file
  - how to read, 2-5
- XPath Support in RTF Templates, 2-119
- XSL:FO elements
  - using in RTF templates, 2-119
- XSL elements
  - apply a template rule, 2-125
  - call template, 2-126
  - copy the current node, 2-126
  - define the root element of the stylesheet, 2-127
  - import stylesheet, 2-126
  - template declaration, 2-126
  - using in RTF templates, 2-125
  - variable declaration, 2-126
  - XML Publisher syntax for , 5-6