Liant Software Corporation

# XML Extensions™

## User's Guide

### First Edition

LIANT

This manual is a user's guide for Liant Software Corporation's XML Extensions, a system designed to allow RM/COBOL applications to access XML documents. It is assumed that the reader has a basic understanding of XML. It is also assumed that the reader is familiar with programming concepts and with the COBOL language in general.

The information contained herein applies to systems running under Microsoft 32-bit Windows and UNIX operating systems

The information in this document is subject to change without prior notice. Liant Software Corporation assumes no responsibility for any errors that may appear in this document. Liant reserves the right to make improvements and/or changes in the products and programs described in this guide at any time without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted.



**Liant Software Corporation**
8911 N. Capital of Texas Highway
Austin, TX 78759
U.S.A.

| | |
|---|---|
| Phone | (512) 343-1010<br>(800) 762-6265 |
| Fax | (512) 343-9487 |
| Web site | http://www.liant.com |

---

## Documentation Release History for the XML Extensions User's Guide:

| Edition Number | Document Part Number | Applies To Product Version | Publication Date |
|---|---|---|---|
| 1 | 401229 | XML Extensions version 9 and later | September 2005 |

# Contents

## List of Tables

# Preface

## Welcome to XML Extensions

XML Extensions for RM/COBOL is Liant Software Corporation's facility that allows RM/COBOL applications to access Extensible Markup Language (XML) documents. XML is the universal format for structured documents and data on the Web. Adding "structure" to documents facilitates searching, sorting, or any one of a variety of operations that can be performed on an electronic document.

**Note** Beginning with the version 9 release, the name of this product changed from XML Toolkit to XML Extensions.

XML Extensions has many capabilities. The major features support the ability to import and export XML documents to and from COBOL working storage. Specifically, XML Extensions allows data to be imported from an XML document by converting data elements (as necessary) and storing the results into a matching COBOL data structure. Similarly, data is exported from a COBOL data structure by converting the COBOL data elements (as necessary) and storing the results in an XML document.

Version 9 of XML Extensions for RM/COBOL runs on Microsoft Windows 32-bit operating systems and selected UNIX platforms. It requires RM/COBOL version 8 or later.

**Note** Microsoft Windows 95 is not supported because the underlying XML parser (Microsoft's MSXML 4.0) is not supported on Windows 95. Certain UNIX platforms are not supported because the underlying XML parser (from the Gnome project) has not yet been ported to those platforms.

# What's New

The following improvements have been incorporated into the version 9 release of XML Extensions for RM/COBOL on Windows and UNIX. Many of these enhancements have also been distributed with various releases of Business Information Server (BIS).

**Note** For information on the significant enhancements in previous releases of XML Extensions, see Appendix D: *Summary of Enhancements* (on page 185). Deficiencies that are version-specific or are discovered after publication are described in the README files contained on the delivered media.

- **RM/COBOL Object Version 12 Support.** XML Extensions now supports RM/COBOL object version 12, which was introduced with RM/COBOL version 9. For more information, see What is the cobtoxml Utility? (on page 57).
- **UNIX Diagnostics.** Better diagnostic information is returned when XML IMPORT FILE/TEXT statements, discussed in Document Processing Statements (on page 64), fail due to an XSLT transform error.
- **Windows XSLT Stylesheet Processing.** XSLT stylesheets that used a literal result element were incorrectly encoded in UTF-16 on Windows. The encoding for the literal result was fixed to be UTF-8.
- **Missing Windows MSXML Parser.** A more descriptive diagnostic is returned if Microsoft's MSXML 4.0 parser is not installed. See System Requirements for Windows (on page 9) for further details.
- **Buffer Overrun Problem.** The XML import statements now verify that input data will fit in selected data structure.
- **URL Recognition.** Previously, only file names that began with "http://" were recognized as URLs. This has been expanded to include "https://".
- **Filename Extensions.** Normally, if a filename extension is not present, one is added. However, with URLs (especially on the Internet), the filename must be used exactly as it is specified. Consequently, the processing of filename extensions has been modified so that a filename extension is never added to a filename that is a URL.
- **RUNPATH Search.** The RUNPATH search sequence has been modified to ignore directory names that use the Universal Naming Convention (UNC) notation (for example, "//system/directory"). UNC names are normally used in an application that uses RM/InfoExpress. XML Extensions cannot access files directly through RM/InfoExpress. By ignoring UNC directory names, unnecessary time delays are avoided when performing a RUNPATH search. For further information, see Automatic Search for Files (on page 37).
- **cobtoxml Banner.** The **cobtoxml** utility has been modified to display the banner when necessary command line parameters are omitted. For more information, see Command Line Interface (on page 58) in Chapter 5: *cobtoxml Utility Reference*.
- **XML Export Blank Suppression.** In prior versions, the XML EXPORT FILE/TEXT statements would strip leading spaces from all non-numeric data items. Leading spaces are now stripped only from data items that are defined with the JUSTIFIED phrase.

- **XSLT Stylesheets with DTD.** The loading of XSLT stylesheets has been improved to allow the stylesheet to contain a document type definition (DTD). Previously, the presence of a DTD in a stylesheet caused a validation error on load. A DTD is required if the stylesheet uses entity references that are not predefined by XML. Stylesheets with HTML or XHTML entity references, such as " " and "©" are often generated by commonly used stylesheet generator tools. The tool may not generate the DTD, so the DTD must be added manually after the XSLT stylesheet is generated. For more information, see Document Type Definition Support (on page 53) in Chapter 4: *XML Considerations*.
- **Improved Namespace Support for Schema Validation.** The XML VALIDATE FILE/TEXT statements would fail in the LoadSchema function if the specified schema contained a targetNamespace attribute. This has been fixed so that the schema loads successfully and is properly referenced in the schema collection by the URL used as the value of the targetNamespace attribute of the schema.

# About Your Documentation

XML Extensions for RM/COBOL documentation consists of a user's guide, which is distributed electronically in Portable Document Format (PDF) as part of the XML Extensions software distribution CD-ROM and at http://www.liant.com/docs.

**Note** To view and print PDF files, you need to install Adobe Acrobat Reader, a free program available from Adobe's Web site at www.adobe.com.

The *XML Extensions User's Guide* is designed to allow you to quickly locate the information you need. The following lists the topics that you will find in the manual and provides a brief description of each.

**Chapter 1—Installation and Introduction.** This chapter describes the installation process and system requirements, and provides a general overview of XML Extensions for RM/COBOL.

**Chapter 2—Getting Started with XML Extensions.** This chapter presents the basic concepts used in XML Extensions for RM/COBOL by creating an example XML-enabled application.

**Chapter 3—COBOL Considerations.** This chapter provides information specific to using RM/COBOL when developing an XML-enabled application.

**Chapter 4—XML Considerations.** This chapter provides information specific to using XML when using XML Extensions with RM/COBOL to develop an XML-enabled application.

**Chapter 5—cobtoxml Utility Reference.** This chapter describes the **cobtoxml** utility (**cobtoxml.exe** on Windows and **cobtoxml** on UNIX) used by XML Extensions and the XML document files, known as model files, that are produced when the **cobtoxml** utility processes the symbol table of a previously compiled RM/COBOL object file.

**Chapter 6—xmlif Library Reference.** This chapter describes the **xmlif** library, a 32-bit dynamic link library on Windows (**xmlif.dll**) or a shared object on UNIX (**xmlif.so**), used by XML Extensions for RM/COBOL.

**Appendix A—XML Extensions Examples.** This appendix contains descriptions of programs or program fragments that illustrate how the **xmlif** library statements are used. These example programs are included with the development system in the XML Extensions examples directory, **Examples**.

**Appendix B—XML Extensions Sample Application Programs.** This appendix provides information about the self-contained XML Extensions sample application programs that are included with the development system in the XML Extensions samples directory, **Samples**.

**Appendix C—XML Extensions Error Messages.** This appendix lists and describes the messages that can be generated during the use of XML Extensions for RM/COBOL.

**Appendix D—Summary of Enhancements.** This appendix reviews the new features and enhancements that were added to earlier releases of XML Extensions.

The *XML Extensions* manual also includes a glossary and an index.

# Related Publications

For additional information, refer to the following publications:

- *RM/COBOL User's Guide*
- *RM/COBOL Language Reference Manual*
- *RM/COBOL Syntax Summary*

# Symbols and Conventions

The following typographic conventions are used throughout this manual to help you understand the text material and to define syntax:

1. Words in all capital letters indicate COBOL reserved words, such as statements, phrases, and clauses; acronyms; configuration keywords; environment variables, and RM/COBOL Compiler and Runtime Command line options.
2. Text that is displayed in a monospaced font indicates user input or system output (according to context as it appears on the screen). This type style is also used for sample command lines, program code and file listing examples, and sample sessions.
3. Bold, lowercase letters represent filenames, directory names, programs, C language keywords, and attributes.

   Words you are instructed to type appear in bold. Bold type style is also used for emphasis, generally in some types of lists.
4. Italic type identifies the titles of other books and names of chapters in this guide, and it is also used occasionally for emphasis.

   In COBOL syntax, italic text denotes a placeholder or variable for information you supply, as described below.

5. The symbols found in the COBOL syntax charts are used as follows:
   a. *italicized words* indicate items for which you substitute a specific value.
   b. UPPERCASE WORDS indicate items that you enter exactly as shown (although not necessarily in uppercase).
   c. ... indicates indefinite repetition of the last item.
   d. | separates alternatives (an either/or choice).
   e. [ ] enclose optional items or parameters.
   f. { } enclose a set of alternatives, one of which is required.
   g. {| |} surround a set of unique alternatives, one or more of which is required, but each alternative may be specified only once; when multiple alternatives are specified, they may be specified in any order.
6. All punctuation must appear exactly as shown.
7. Key combinations are connected by a plus sign (+), for example, Ctrl+X. This notation indicates that you press and hold down the first key while you press the second key. For example, "press Ctrl+X" means to press and hold down the Ctrl key while pressing the X key. Then release both keys.
8. The term "Windows" in this document refers to 32-bit Microsoft Windows operating systems, excluding Windows 95.
9. RM/COBOL Compile and Runtime Command line options may be preceded by a hyphen. If any option on a command line is preceded by a hyphen, then a leading hyphen is required for all options. When assigning a value to an option, the equal sign is optional if leading hyphens are used.

# Registration

Please take a moment to fill out and mail (or fax) the registration card you received with RM/COBOL. You can also complete this process by registering your Liant product online at: http://www.liant.com.

Registering your product entitles you to the following benefits:

- **Customer support.** Free 30-day telephone support, including direct access to support personnel and 24-hour message service.
- **Special upgrades.** Free media updates and upgrades within 60 days of purchase.
- **Product information.** Notification of upgrades, revisions, and enhancements as soon as they are released, as well as news about other product developments.

You can also receive up-to-date information about Liant and all its products via our Web site. Check back often for updated content.

# Technical Support

Liant Software Corporation is dedicated to helping you achieve the highest possible performance from the RM/COBOL family of products. The technical support staff is committed to providing you prompt and professional service when you have problems or questions about your Liant products.

These technical support services are subject to Liant's prices, terms, and conditions in place at the time the service is requested.

While it is not possible to maintain and support specific releases of all software indefinitely, we offer priority support for the most current release of each product. For customers who elect not to upgrade to the most current release of the products, support is provided on a limited basis, as time and resources allow.

## Support Guidelines

When you need assistance, you can expedite your call by having the following information available for the technical support representative:

1. Company name and contact information.
2. Liant product serial number (found on the media label, registration card, or product banner message).
3. Product version number.
4. Operating system and version number.
5. Hardware, related equipment, and terminal type.
6. Exact message appearing on screen.
7. Concise explanation of the problem and process involved when the problem occurred.

## Test Cases

You may be asked for an example (test case) that demonstrates the problem. Please remember the following guidelines when submitting a test case:

- The smaller the test case is, the faster we will be able to isolate the cause of the problem.
- Do not send full applications.
- Reduce the test case to one or two programs and as few data files as possible.
- If you have very large data files, write a small program to read in your current data files and to create new data files with as few records as necessary to reproduce the problem.
- Test the test case before sending it to us to ensure that you have included all the necessary components to recompile and run the test case. You may need to include an RM/COBOL configuration file.

When submitting your test case, please include the following items:

1. **README text file that explains the problems.** This file must include information regarding the hardware, operating system, and versions of all relevant software (including the operating system and all Liant products). It must also include step-by-step instructions to reproduce the behavior.
2. **Program source files.** We require source for any program that is called during the course of the test case. Be sure to include any copy files necessary for recompilation.
3. **Data files required by the programs.** These files should be as small as possible to reproduce the problem described in the test case.

# Chapter 1: Installation and Introduction

This chapter describes the system requirements and installation processes for development and deployment on both Windows and UNIX operating systems. It also provides a general overview of XML Extensions for RM/COBOL and the benefits it offers to the COBOL programmer.

**Note** You should have a basic understanding of XML in order to use XML Extensions. Depending on the complexity of your application, you may also need to know about XSLT stylesheets.

## Before You Start

Before you follow the instructions for installing XML Extensions for RM/COBOL (on page 11), make sure that your computer configuration meets the following minimum hardware and software requirements for each of the supported architectures, and that your XML Extensions package contains the necessary items for development and deployment.

**Note** You may wish to use Microsoft Internet Explorer, version 6 or greater, as a convenient tool for viewing XML documents.

### System Requirements

To run XML Extensions for RM/COBOL, you must have certain hardware and software installed on your computer.

#### *For Windows*

The system requirements for Windows include the following:

- The XML Extensions hardware and software requirements are the same as RM/COBOL version 9 for 32-bit Windows. (See the *RM/COBOL User's Guide*, First Edition or later.) Additionally, XML Extensions may be used in conjunction with Terminal Server.

- Microsoft's XML parser, MSXML 4.0 or greater, is also required. (A schema processor and an XSLT transformation processor are included in the Microsoft MSXML 4.0 parser.)

  **Note** The MSXML 4.0 parser may fail to install correctly if the target system does not have either Microsoft Windows Installer or Internet Explorer installed. Both of these products are freely available from Microsoft. To obtain these applications, follow the www.microsoft.com/downloads/search.asp link and search for the keywords "windows installer 2.0" or "internet explorer", as needed.

### *For UNIX*

The system requirements for UNIX include the following:

- The XML Extensions hardware and software requirements are the same as RM/COBOL version 9 for UNIX. (See the *RM/COBOL User's Guide*, First Edition or later.)

  **Note 1** The XML parser (libxml) and the XSLT transformation processor (libxslt) from the C libraries for the Gnome project are included in XML Extensions.

  **Note 2** While the Windows implementation continues to support the use of schema files (on page 55), the UNIX implementation does not currently support this capability. Schema support in the underlying XML parser (libxml) is still under development.

## XML Extensions Package

The XML Extensions for RM/COBOL package contains the following items for development and deployment.

### *Development*

The XML Extensions development system includes the following files:

- Deployment files. These files are listed in the Deployment (on page 11) section.
- **cobtoxml** command line utility (**cobtoxml.exe** on Windows and **cobtoxml** on UNIX). For more information, see Chapter 5: *cobtoxml Utility Reference* (on page 57).
- XML document files used by the **cobtoxml** utility (**toxdr.xsl**, **toxdrb.xsl**, **toxsd.xsl**, and **toxsl.xsl**).
- Copy files (**lixmlall.cpy**, **lixmldef.cpy**, **lixmldsp.cpy**, **lixmlrpl.cpy**, and **lixmltrm.cpy**). For more details, see Copy Files (on page 46).
- Example files. These programs or program fragments illustrate how **xmlif** library statements are used. For further information, see Appendix A: *XML Extensions Examples* (on page 87). The example programs can be found in the XML Extensions example directory, **Examples**.
- Sample files. These self-contained, working application programs, which include the complete source, can be used in your own applications by modifying or customizing them, as necessary. See Appendix B: *XML Extensions Sample Application Programs* (on page 175) for more details. The sample application programs can be found in the XML Extensions sample directory, **Samples**.

### *Deployment*

The XML Extensions deployment system consists of the following files:

- **xmlif** COBOL-callable subprogram library (**xmlif.dll** on Windows and **xmlif.so** on UNIX). For more information, see Chapter 6: *xmlif Library Reference* (on page 63).
- For Windows, MSXML 4.0, the Microsoft XML parser, schema processor, and XSLT transformation processor (**msxml4.dll**, **msxml4a.dll**, and **msxml4r.dll**).

  For UNIX, the XML parser and XSLT transformation processor libraries (**libxml** and **libxslt**, respectively). Currently, these libraries are linked into the **xmlif.so** file and do not need to be installed separately.

The developer should deploy the model files that were generated by the **cobtoxml** utility along with the COBOL program files. Normally, these files are stored in the same location as the COBOL program files. For more information, see Model Files (on page 28).

# Installing XML Extensions

The following sections describe the distribution media options, and how to install the XML Extensions for RM/COBOL development and deployment systems on Windows (see page 12) and UNIX (see page 14). XML Extensions is available as a development system and a deployment system. The development system is designed to operate in conjunction with an RM/COBOL development system. The deployment system is designed to operate in conjunction with an RM/COBOL runtime system.

For development, both the XML Extensions development system and Liant's RM/COBOL version 9 development system are required. For deployment, both the XML Extensions deployment system and the RM/COBOL version 9 runtime system are required.

## Distribution Media Options

The XML Extensions for RM/COBOL software is available on CD-ROM media and via Electronic Software Delivery.

Electronic Software Delivery allows XML Extensions to be downloaded from the Liant Electronic Software Delivery Web site in any of the following three formats. Simply follow the instructions on the Web site for downloading and decompressing the file for the format selected:

- **Windows Self-Extracting EXE**. The download format for Windows Self-Extracting EXE does *not* contain a full product. In order to reduce the size of the download file, the AutoPlay and Adobe Acrobat Reader software, normally available on the CD, are not present. Furthermore, the *XML Extensions* manual is included in the deliverable download file but the installation script to install the manual is not. The *XML Extensions* manual, formatted as a PDF file, can be installed simply by copying it to the desired location. After the deliverable has been downloaded and uncompressed, and the installation components have been created from the Windows Self-Extracting EXE format, follow the instructions under Installing on Windows (on page 12).

- **UNIX ISO CD Image**. The download format for ISO CD Image contains the full product. Use CD-ROM Burning software, such as Nero (http://www.nero.com) or Roxio's Easy CD Creator (http://www.roxio.com), to create the physical CD-ROM media. See the instructions for loading the distribution media (on page 17) on specific versions of UNIX. After the CD is created, proceed with normal CD installation as described in Performing the Installation (on page 18).
- **UNIX GUNZIP TAR**. The download format for UNIX GUNZIP TAR download does *not* contain a full product. In order to reduce the size of the download file, the AutoPlay and Adobe Acrobat Reader software, normally available on the CD, are not present. Furthermore, the *XML Extensions* manual is included in the deliverable download file but the installation script to install the manual is not. The *XML Extensions* manual, formatted as a PDF file, can be installed simply by copying it to the desired location.

  **Note** If the distribution was via Electronic Software Delivery using UNIX GUNZIP TAR, after the downloaded file has been decompressed into an installation components directory, no further steps are necessary to load the distribution media.

## Installing on Windows

This section contains instructions on how to install the XML Extensions for RM/COBOL development system and deployment system on Windows.

### *Install the Development System on Windows*

To install the XML Extensions for RM/COBOL development system (**XMLEXT_D.EXE**) on Windows:

1. Restart Windows. (Be sure not do not start any other applications.)
2. Insert the XML Extensions for RM/COBOL Development Installation CD into your CD-ROM drive.

   The installation program starts automatically.
3. Click **I Agree** to accept the license agreement.
4. In the Installation Options dialog box, select **XML Examples** (if desired), and then click **Next** to continue.
5. In the Installation Directory dialog box, accept the location presented or click **Browse** to select another location.

   **Note** The installation automatically locates the RM/COBOL development system and selects this directory as the default location for the XML Extensions development system installation.
6. Click **Install** to continue.
7. When the Liant License File dialog box opens, insert the license diskette that accompanied the installation CD in the diskette drive in your computer.
8. Enter the complete file name for the Liant license file. The default name is a:\liant.lic.

   **Note** If you are using a drive other than A, be sure to correct the location of the license file in the Liant License File dialog box. If necessary, the **liant.lic**

license file can be copied to a location on a hard drive and that location can be specified during installation.

9. Click **Next** to continue.
10. When the installation completes, the Completion dialog box is displayed. Click **Close** to dismiss this dialog box.
11. At the "Setup has completed. View readme file now?" prompt, do one of the following:
    - Select **Yes** to view the README file.
    - Select **No** to open the XML Extensions for RM/COBOL window.

## *Install the Deployment System on Windows*

To install the XML Extensions for RM/COBOL deployment system (**XMLEXT_R.EXE**) on Windows:

**Note** The XML Extensions deployment system is provided as a self-extracting executable that installs the deployment system components of XML Extensions. It is delivered on the XML Extensions Development Installation CD as **redist\XMLEXT_R.EXE**.

1. Start Windows.
2. Insert the XML Extensions for RM/COBOL Development Installation CD into your CD-ROM drive.
3. When the **XMLEXT_R.EXE** file is executed, the Installation Directory dialog box is displayed.
4. In the Installation Directory dialog box, accept the location presented or click **Browse** to select another location.

   The installation program automatically locates and selects the RM/COBOL runtime system directory as the default location for the XML Extensions deployment system installation.
5. Click **Install** to continue.
6. When the installation completes, the Completion dialog box is displayed. Click **Close** to dismiss the dialog box.

**Notes**

- Your license for this product does not allow you to redistribute the entire XML Extensions development system with your application. You may redistribute only the deployment system.
- Provide the file, **XMLEXT_R.EXE**, to your end-users along with your application. Either package this file in an installation process so that it is executed on the target platform or instruct your end-users to execute the file once on their system to install the necessary components as part of setting up the application.

# Installing on UNIX

This section contains instructions on how to install the XML Extensions for RM/COBOL development system and deployment system on UNIX.

## *Install the Development System on UNIX*

XML Extensions for RM/COBOL is delivered on media suitable for your configuration. In order to use this product, you must install the contents of the distribution media onto your hard disk.

There are four main steps to installing the XML Extensions development system for UNIX:

1. Loading the license file (see the following topic).
2. Loading the distribution media (on page 17).
3. Performing the installation (on page 18).
4. Unloading the distribution media (on page 18).

### Loading the License File

The Liant license file is a normal text file distributed on an MS-DOS-formatted diskette. Not all UNIX operating systems, however, can read an MS-DOS-formatted diskette, and not all UNIX server machines have diskette drives. To make the license file available to the RM/COBOL for UNIX installation script, two techniques are provided:

1. Mount the diskette as an MS-DOS file system.
2. Transfer the Liant license file via FTP from a Windows client.

#### *Mount the Diskette as an MS-DOS File System*

Use this option to load the license file if the UNIX operating system supports MS-DOS file systems and your hardware has a diskette drive installed. Instructions for specific platforms and versions of UNIX are provided.

- **Digital UNIX (Tru64)**, **HP-UX**, and **IBM AIX**, and **Intel UNIX System V Release 4.** These platforms do not support mounting MS-DOS diskettes. Use the FTP instructions on page 16 to transfer the license file to the UNIX server.
- **Linux**

  a. Insert the diskette into the diskette drive.

  b. Log in as root and enter:

  ```
  mount –t msdos /dev/fd0H1440 /mnt/floppy
  ```

  c. Copy the license file to the /tmp directory:

  ```
  cp /mnt/floppy/liant.lic /tmp/liant.lic
  ```

  d. Dismount the diskette with the following command and then remove the diskette from the diskette drive:

  ```
  umount /mnt/floppy
  ```

- **SCO OpenServer 5**

  a. Insert the diskette into the diskette drive.

  b. Log in as root and enter:

  ```
  mount –f DOS,lower /dev/fd0 /floppy
  ```

  **Note** It may be necessary to create the mount directory, /floppy, before executing this command.

  c. Copy the license file to the /tmp directory:

  ```
  cp /floppy/liant.lic /tmp/liant.lic
  ```

  d. Dismount the diskette with the following command and then remove the diskette from the diskette drive:

  ```
  umount /floppy
  ```

- **SCO UnixWare 7 and OpenServer 6 (SCO SVR5)**

  a. Insert the diskette into the diskette drive.

  b. Log in as root and enter:

  ```
  mount –F dosfs /dev/dsk/f0q18dt /Disk_A
  ```

  c. Copy the license file to the /tmp directory:

  ```
  cp /Disk_A/liant.lic /tmp/liant.lic
  ```

  d. Dismount the diskette with the following command and then remove the diskette from the diskette drive:

  ```
  umount /Disk_A
  ```

- **Sun Solaris**

  a. Insert the diskette into the diskette drive.

  b. Log in as root and enter:

  ```
  volcheck
  ```

  c. Copy the license file to the /tmp directory:

  ```
  cp /floppy/liant/liant.lic /tmp/liant.lic
  ```

  d. Dismount the diskette with the following command and then remove the diskette from the diskette drive:

  ```
  eject floppy
  ```

### *Transfer the Liant License File via FTP from a Windows Client*

To transfer the Liant license file from a Windows client to the UNIX server, use one of the many graphical FTP utilities available on Windows and transfer the **liant.lic** license file as a text file. You can also follow this procedure:

1. On the Windows client, insert the diskette into the diskette drive.

   These instructions assume that this is drive A. If it is another drive, change the drive letter to the appropriate letter in the remaining instructions.

2. Open an MS-DOS Prompt window by clicking Start on the task bar, point to Programs, and then click MS-DOS Prompt.

3. Connect to the UNIX server by entering:

   ```
   ftp UnixServerName
   ```

   where *UnixServerName* is the network name of your UNIX server.

4. Change the directory to the /tmp directory:

   ```
   cd /tmp
   ```

5. Specify a text file transfer:

   ```
   ASCII
   ```

6. Send the license file to the UNIX server:

   ```
   send A:\LIANT.LIC liant.lic
   ```

7. Disconnect from the UNIX server:

   ```
   bye
   ```

8. Close the MS-DOS Prompt window with the following command and then remove the diskette form the diskette drive:

   ```
   Exit
   ```

## Loading the Distribution Media

To load the distribution media on the UNIX machine:

1. Insert the RM/COBOL for UNIX CD-ROM in the appropriate CD-ROM drive.
2. Log in as root.
3. Enter the appropriate mount command for your system. See the examples listed below.

**Note 1** In the list that follows, the standard mount directory names are used where the UNIX operating system has such a standard. If the operating system does not follow a standard, the name */cdrom* is used. In such cases, it will be necessary either to create the directory */cdrom* or to substitute the preferred mount directory name for */cdrom*.

**Note 2** The device names below are examples. The actual device name is dependent on the hardware configuration of your UNIX server. It may be necessary to substitute the proper value for your system. Consult your UNIX System Administrator for more details.

| System | Mount Command |
|---|---|
| Digital UNIX (Tru64) | `mount -t cdfs -o ro,noversion /dev/rz4c /cdrom` |
| HP-UX | `mount –F cdfs –o ro,cdcase /dev/dsk/c0t4d0 /cdrom` |
| IBM AIX | `mount –o ro –v cdrfs /dev/cd0 /cdrom` |
| Intel UNIX System V Release 4 | `mount –o ro –F cdfs /dev/cdrom/c0t4l0 /cdrom` |
| Linux | `mount –o ro –t iso9660 /dev/cdrom /mnt/cdrom` |
| SCO OpenServer 5 | `mount –o ro –f ISO9660,lower /dev/cd0 /cdrom` |
| SCO UnixWare 7 and OpenServer 6 (SCO SVR5) | `mount –F cdfs –o ro /dev/cdrom/c1b0t0l0 /CD-ROM_1` |
| Sun Solaris | If Solaris does not automatically load the CD-ROM, log in as root and enter: `volcheck` |

### Performing the Installation

After the CD-ROM has been successfully mounted, you will need to do the following:

1. Change the directory to the mount point for the CD-ROM. For example, enter:

   ```
   cd /cdrom
   ```

2. From the mount point, execute the installation script using the following command:

   ```
   sh ./install.sh
   ```

3. The installation script prompts you for all the information that it needs before beginning the actual installation. Answer the prompts of the installation script accordingly.

Messages are displayed periodically indicating the status of the installation.

**Note** The development and deployment systems are installed separately. The installation script will present you with a prompt to install: A: Development System or B: Deployment System. You choose the one you wish to install.

### Unloading the Distribution Media

To unload (remove) the distribution media from the hardware:

1. Enter the appropriate command for your system. See the examples listed below.
2. Remove the distribution media from the CD-ROM drive.

| System | Umount Command |
|---|---|
| Digital UNIX (Tru64)<br>HP-UX<br>IBM AIX<br>Intel UNIX System V Release 4<br>SCO OpenServer 5 | `umount /cdrom` |
| Linux | `umount /mnt/cdrom` |
| SCO UnixWare 7 and<br>OpenServer 6 (SCO SVR5) | `umount /CD-ROM_1` |
| Sun Solaris | `eject cdrom` |

## *Install the Deployment System on UNIX*

For deploying COBOL applications that use XML Extensions, install the XML Extensions deployment system on each platform that runs the application. You may do this using the XML Extensions installation disk.

# Introducing XML Extensions

XML Extensions for RM/COBOL allows RM/COBOL applications to interoperate freely and easily with other applications that use the Extensible Markup Language (XML) standard. To accomplish this, XML Extensions leverages the similarities between the COBOL data model and the XML data model in order to turn RM/COBOL into an "XML engine." Of primary importance to this goal is the ability to import and export XML documents to and from standard COBOL data structures.

**Note** A COBOL data structure is a COBOL data item. In general, it is a group data item, but in some cases, it may be a single elementary data item. The **cobtoxml** utility, a component of XML Extensions that is described in Chapter 5: *cobtoxml Utility Reference* (on page 57), captures the COBOL data structure, including transformed data-names of the data items and subordinate data items, if any, so that a mapping between the actual COBOL data structure and an XML representation of the COBOL data structure can be accomplished in either direction at runtime.

By allowing standard COBOL data structures to be imported from and exported to XML documents, XML Extensions enables the direct processing and manipulation of XML-based electronic documents by the RM/COBOL application programmer. Furthermore, XML Extensions does this without requiring the application programmer to become thoroughly familiar with the numerous XML-related specifications and the time-consuming process required to emit and consume well-formed XML.

Specifically, an XML document may be imported into a COBOL data structure under COBOL program control using a single, simple COBOL statement, and, similarly, the content of a COBOL data structure may be used to generate an XML document with equal simplicity. XML Extensions' approach handles both simple and extremely complex structures with ease. Individual data elements are automatically converted as needed between their COBOL internal data types and the external coding used by XML. Not only can the transition to and from XML take place when this happens, but powerful transforms, which are coded using Extensible Stylesheet Language Transformations (XSLT), can be applied at the same time. This powerful mechanism gives XML Extensions the capabilities needed to be useful in a wide range of e-commerce and Web applications.

In order to add this powerful document-handling capability to a COBOL application, the programmer need only describe the information to be received or transmitted to the external components as COBOL data definitions. In many cases, this description will simply be the already-existing data area defined in the COBOL application. Once the "document" content is described in this way, a simple command-line utility program (**cobtoxml.exe** on Windows and **cobtoxml** on UNIX), referenced throughout this document as the **cobtoxml** utility, is run, specifying the data structures to be "opened" to the XML world. This utility captures all the information needed in a set of XML documents called model files. At application execution time, a COBOL statement, accessed via a library of statements defined in copy files supplied with XML Extensions, is used to call a subprogram which implements the complete runtime functionality of XML Extensions. This COBOL-callable subprogram library (**xmlif.dll** on Windows and **xmlif.so** on UNIX) is referenced throughout this document as the **xmlif** library. For more information, see Chapter 5: *cobtoxml Utility Reference* (on page 57), and Chapter 6: *xmlif Library Reference* (on page 63).

## What is XML?

In this document, XML refers to the entire set of specifications and products related to a particular approach to representing structured information in text-based form. Specifically, the World Wide Web Consortium (W3C) has specified a markup-based language called XML. Closely related to HTML, XML was designed to build on what had been learned with that, now ten-year-old, technology. Among other things, XML was designed to be much more generally useful than HTML, while exhibiting the simplest possible expression. HTML is about displaying information. It was designed to display data and to focus on how the data looks. XML, meanwhile, is about describing information. It was designed to describe data and focus on what the data is. Since XML's definition, a constellation of XML-related specifications has been produced and is in progress to leverage the power of this new form of information expression.

For the COBOL programmer, it is best to view XML not as a markup language for text documents, but rather as a text-based encoding of a general abstract data model. It is this data model, and its similarity to COBOL's data model, that yields its power as an adjunct to new and legacy COBOL applications needing to interact with other applications and systems in the most modern way possible.

XML is extremely important to the COBOL programmer for two key reasons. First, it is rapidly becoming the standard way of exchanging information on the Web, and second, the nearly perfect alignment of the COBOL way of manipulating data and the XML information model results in COBOL being arguably the best possible language for expressing business data processing functions in an XML-connected world.

### *COBOL as XML*

What does XML look like? Start with the assumption that it is a textual encoding of COBOL data (although this is not quite accurate, it is sufficient for now). Suppose you have the following COBOL definition in the Working-Storage Section:

```
01 contact.
   10 firstname pic x(10) value "John".
   10 lastname pic x(10) value "Doe".
   10 address.
      20 streetaddress pic x(20) value "1234 Elm Street".
      20 city pic x(20) value "Smallville".
      20 state pic x(2) value "TX".
      20 postalcode pic 9(5) value "78759".
   10 email pic x(20) value "jd@aol.com".
```

What does this information look like if you simply WRITE it out to a text file? It looks like this:

```
John      Doe       1234 Elm Street     Smallville          TX78759jd@aol.com
```

You can see that all the "data" is here, but the "information" is not. If you received this, or tried to read the file and make sense out of it, you would need to know more about the data. Specifically, you would have to know how it is structured and the sizes of the fields. It would be helpful to know how the author named the various fields as well, since that would probably give you a clue as to the content.

This is not a new problem; it is one that COBOL programmers (as well as other application programmers) have had to deal with on an ad hoc basis since the

beginning of the computer age. But now, XML gives us a way to encode all of the information in a generally understandable way.

Here is how this information would be displayed in an XML document:

```
<contact>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
    <address>
       <streetaddress>1234 Elm Street</streetaddress>
       <city>Smallville</city>
       <state>TX</state>
       <postalcode>78759</postalcode>
    <email>jd@aol.com</email>
</contact>
```

In XML, the COBOL group-level item is coded in what is called an "element." Elements have names, and they contain both text and other elements. As you can see, an XML element corresponds to a COBOL data item. In this case, the 01-level item "contact" becomes the <contact> element, coded as a start "tag" ("<contact>") and an end tag ("</contact>") with everything in between representing its "content." In this case, the <contact> element has as its content the elements <firstname>, <lastname>, <address>, and <email>. This corresponds precisely to the COBOL Data Division declaration for "contact." Similarly, the 10-level group item, "address", becomes the element <address>, made up of the elements <streetaddress>, <city>, <state>, and <postalcode>. Each of the COBOL elementary items is coded with text content alone. Notice that in the XML form, much of the semantic information is missing from the raw COBOL output form of the data. As a bonus, you no longer have the extraneous trailing spaces in the COBOL elementary items, so they are removed. In other words, the XML version of this record contains both the data itself and the structure of the data.

Now, what if the COBOL data had looked like the following:

```
01 contact.
   10 email pic x(20)
   10 firstname pic x(10).
   10 lastname pic x(10).
   10 address.
      20 city pic x(20).
      20 state pic x(2).
      20 postalcode pic 9(5).
      20 streetaddresslines pic 9.
      20 streetaddresses.
         30 streetaddresses occurs 1 to 9 times
             depending on streetaddresslines pic x(20).
```

Two things have changed in this example: the initial values have been removed and there can now be up to nine “streetaddress” items. This is much more similar to what you might expect in a real application. After the application code sets the values of the various items from the Procedure Division, the XML coding of the result might look like this:

```
<contact>
    <email>bs@aol.com</email>
    <firstname>Betty</firstname>
    <lastname>Smith</lastname>
    <address>
          <city>Galesburg</city>
          <state>IL</state>
          <postalcode>61401</postalcode>
          <streetaddresslines>3</streetaddresslines>
          <streetaddresses>
           <streetaddress>Knox College</streetaddress>
           <streetaddress>Campus Box 9999</streetaddress>
           <streetaddress>2 E. South St.</streetaddress>
          </streetaddresses>
</contact>
```

Notice the repeating item “streetaddress” has become three <streetaddress> elements. In this example, COBOL acts as an XML programming language, providing both the structure (schema) of the data and the data itself.

Even though these examples are very simple, they illustrate how powerful the compatibility between the COBOL data model and the XML information model can be. COBOL structures of arbitrary complexity have a straightforward XML representation. There are, it turns out, some things that you can specify in a COBOL data definition that cannot be coded as XML, but these can easily be avoided if you are programming your application for XML.

## *XML as COBOL*

In the previous cases, you saw how structured COBOL data could be coded as an XML document. In this section, you will examine how an arbitrary XML document can be represented as a COBOL structure. This requires that you look at some other aspects of the XML information model that are not needed to represent COBOL structures, but might be present in XML, nonetheless.

So far, you have seen that XML has elements and text. Although, these are the primary means of representing data in XML documents, there are some other ways of representing and structuring data in XML. Suppose you have the following XML document:

```
<contact type="student">
    <firstname>Betty</firstname>
    <lastname>Smith</lastname>
    <address form="US">
          <streetaddresses>
                <streetaddress>Knox College</streetaddress>
                <streetaddress>Campus Box 9999</streetaddress>
                <streetaddress>2 E. South St.</streetaddress>
          </streetaddresses>
          <city>Galesburg</city>
          <state>IL</state>
          <postalcode zipplus4="N">61401</postalcode>
    <email>bs@aol.com</email>
</contact>
```

In the example document shown here is now a new kind of data, known as an "attribute" in XML. Notice that the <contact> element tag has what appears to be some kind of parameter named "type." This is, in fact, an attribute whose value is set to the text string "student." In XML, attributes are another way of coding element content, but in a way that does not affect the text content of the element itself. In other words, attributes are "out-of-band" data associated with an element. This concept has no parallel in standard COBOL. In COBOL, all data associated with a data item is part of the COBOL record content. This means that if you are to capture all of the content of an XML document, you must have a way to capture and store attributes.

You do this with the help of an important XML tool called an external XSLT stylesheet file (see page 55). (In this document, "external XSLT stylesheet" is used to differentiate an XSLT stylesheet provided by the user from the "internal XSLT stylesheet" generated as one of the model files by the **cobtoxml** utility and used automatically as part of importing XML documents into COBOL.) For now, assume that an XSLT stylesheet can transform an XML document into any desired alternative XML document. If this is true (and it is), you must code the incoming attributes as something that has a direct COBOL counterpart. This would be as a data item represented as a text element in XML.

The example document, after external XSLT stylesheet transformation, might look like this:

```
<contact>
    <email>bs@aol.com</email>
    <attr-type>student</attr-type>
       <firstname>Betty</firstname>
       <lastname>Smith</lastname>
       <address>
              <attr-form>US</attr-form>
              <city>Galesburg</city>
              <state>IL</state>
              <postalcodegroup>
                     <attr-zipplus4>N</attr-zipplus4>
                     <postalcode>61401</postalcode>
              </postalcodegroup>
    <streetaddresslines>3</streetaddresslines>
              <streetaddresses>
                     <streetaddress>Knox College</streetaddress>
                     <streetaddress>Campus Box 9999</streetaddress>
                     <streetaddress>2 E. South St.</streetaddress>
               </streetaddresses>
</contact>
```

Several things have been changed. The attributes have been turned into elements, but with a special name prefixed by “attr-“ and a new element, <streetaddresslines> has been added containing a count of the number of <streetaddress> elements. In the case of <postalcode>, a new element has been added to wrap both the real <postalcode> value, and the new attribute. All of these changes are very easy to make using a simple XSLT stylesheet, and you now have a document with a direct equivalent in COBOL:

```
01 contact.
   10 email pic x(20).
   10 attr-type pic x(7).
   10 firstname pic x(10).
   10 lastname pic x(10).
   10 address.
      20 city pic x(20).
      20 state pic x(2).
      20 postalcodegroup.
         30 attr-zipplus4 pic x.
         30 postalcode pic 9(5).
      20 attr-form pic xx.
      20 streetaddresslines pic 9.
      20 streetaddresses.
         30 streetaddress occurs 1 to 9 times
               depending on streetaddresslines pic x(20).
```

# Chapter 2: Getting Started with XML Extensions

This chapter presents the basic concepts used in XML Extensions for RM/COBOL by creating an example XML-enabled application. It also discusses how XML Extensions locates files.

## Overview

Because the COBOL information model can largely be expressed by the XML information model, there is a natural relationship between XML documents and COBOL data structures. Both present similar views of the data; that is, the entire data is visible. You may view the content of a COBOL data record and you may view the text of an XML document. In XML, markup is used both to name and describe the text elements of a document. In COBOL, the data structure itself provides names and descriptions of the elements within a document.

XML Extensions has many capabilities. The major features support the ability to import and export XML documents to and from a COBOL program's Data Division. Note that data may be anywhere in the Data Division. Specifically, XML Extensions allows data to be imported from an XML document by converting data elements, as necessary, and storing the results into a matching COBOL data structure. Similarly, data is exported from a COBOL data structure by converting the COBOL data elements, as necessary, and storing the results in an XML document.

XML Extensions consists of the following two main components:

- The **cobtoxml** utility (**cobtoxml.exe** on Windows and **cobtoxml** on UNIX), which runs as a post-compile step. This program creates a set of XML documents, called model files (on page 28), which describe a selected COBOL data structure as a set of XML documents.
- The **xmlif** library (**xmlif.dll** on Windows and **xmlif.so** on UNIX), which is a COBOL-callable runtime library used to implement a series of COBOL statements that are available to the developer for directing the importing and exporting of COBOL data as XML.

# Typical Development Process Example

This section provides an example of how to produce an XML-enabled application. These instructions assume that both the XML Extensions for RM/COBOL development system and the RM/COBOL development system (version 8 or later) are installed on your computer.

**Note** More examples and information about complete sample application programs can be found in:

- Appendix A: *XML Extensions Examples* (on page 87)
- Appendix B: *XML Extensions Sample Application Programs* (on page 175)
- The XML Extensions examples and samples directories (**Examples** and **Samples**, respectively)

There are five basic steps to developing an XML-enabled application:

1. Design the COBOL data structure (on page 27). Develop a COBOL program, or modify an existing one, using statements that refer to the **xmlif** library.
2. Compile the program (on page 27). Compile the COBOL program with the RM/COBOL Compile Command Y Option enabled in order to place the symbol table in the object file.
3. Run the cobtoxml utility (on page 27). Run the **cobtoxml** utility in order to generate a set of XML model files that describe a data structure within the COBOL program.
4. Execute the COBOL program (on page 30). Test the program and repeat steps 1 through 4, as necessary.
5. Deploy the application (on page 35). After stripping the symbol table information from the COBOL object program, distribute the XML Extensions deployable files. These files consist of the **xmlif** library and the underlying XML parser that this library uses.

The sections that follow describe each of the basic steps involved in the example provided, and they include explanations of how more functionality is added to the program.

## Design the COBOL Data Structure

The first step is to design a COBOL data structure that describes the data to be placed in a corresponding XML document. The following simple example illustrates this step using typical mailing address information. An adequate program skeleton has been included to allow the program to compile without error.

```
Identification Division.
Program-Id.  Getting-Started.
Data Division.
Working-Storage Section.
01  Customer-Address.
    02  Name          Pic X(128).
    02  Address-1     Pic X(128).
    02  Address-2     Pic X(128).
    02  Address-3.
        03  City      Pic X(64).
        03  State     Pic X(2).
        03  Zip       Pic 9(5) Binary.
```

This COBOL data structure contains only one numeric element: the zip code. For demonstration purposes, it is represented as binary.

## Compile the Program

In the second step, you compile the program with the following command line:

```
rmcobol getstarted y
```

This compilation uses the Y Compile Command Option to provide a symbol table in the COBOL object, which is required by the **cobtoxml** utility.

## Run the cobtoxml Utility

The third step is to execute the **cobtoxml** utility from the command line by entering:

```
cobtoxml getstarted customer-address
```

The first parameter, `getstarted`, is the name of the COBOL object file. An extension of **.cob** is automatically assumed, if no extension is provided. The second parameter is the name of the data structure that will be used by the runtime components of XML Extensions.

When the **cobtoxml** utility is run, it generates a set of XML model files that describe a data structure within the COBOL program. The following section describes each of these model files and provides examples.

## *Model Files*

The **cobtoxml** utility creates a set of files that are XML documents, known as model files. Model files have the same root name as the object file, although each filename has a unique, predetermined extension. In this case, the following types of model files are created:

- Example file (**getstarted.xml**)
- Template file (**getstarted.xtl**)
- Internal XSLT stylesheet file (**getstarted.xsl**)
- Schema file (**getstarted.xsd**)

  **Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

The XML model files created in this example are described in more detail in the following sections. See also Referencing XML Model Files (on page 61).

### Example File

The XML document, **getstarted.xml**, is an example file created primarily as a reference for the COBOL developer. It illustrates the form that the COBOL data structure will take when encoded as an XML document. No actual data content is included and the **xmlif** library does not use this file. You may use Microsoft Internet Explorer to view this XML document, which looks like the following. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
 <customer-address>
  <name />
  <address-1 />
  <address-2 />
  <address-3>
   <city />
   <state />
   <zip />
  </address-3>
 </customer-address>
</root>
```

Even if you are not familiar with XML, it is easy to see how the XML document is derived. XML is a markup language—a set of rules (you may also think of them as guidelines or conventions) for designing text formats that let you structure your data. In that way, it is similar to HTML. Markup is descriptive information inserted in the text of a document. Like HTML, XML makes use of tags (words bracketed by '<' and '>') and attributes (of the form `name="value"`).

Nesting of elements is done by using a matched set of beginning (start tags) and ending (end tags) markup. In this example, `<root>` marks a beginning and `</root>` marks an ending. The tags `<customer-address>` and `<address-3>` also both have start tags and end tags. In addition, XML allows a shortcut notation that may be used when a start tag is immediately followed by an end tag (that is, when there is no intervening content). This is known as an “empty element.” The end tag may be omitted by terminating the start tag with the “/>”

sequence. In this example, `<name />` is shorthand for the `<name></name>` sequence. The meaning of both forms is the same, and they can be used interchangeably. Microsoft Internet Explorer recognizes an end tag immediately following a start tag and displays the shorthand instead of the longer version. If you use a text editor (such as Notepad) instead of Internet Explorer to view this XML document, you will observe that the shorthand sequence is not used by an example file.

Since this XML document is intended simply as a reference for the programmer, it contains no text, only markup. Notice that the first line is an XML header, which is always generated. The `<root>` tag also is always generated. Nested inside the root element is the `customer-address` element. This was generated from the `customer-address` data-name in the COBOL program. Because names in XML are case sensitive and names in COBOL are case insensitive, the name in the COBOL program is converted to all lowercase for consistency.

## Template File

The XML document, **getstarted.xtl**, is a template file that is used by the **xmlif** library when exporting a document (converting from COBOL to XML). It is similar to the example file, but it includes much more information. This document contains XML attributes in addition to elements. The attributes provide the additional information the **xmlif** library needs to encode the COBOL data properly as XML at runtime.

Attributes are associated with an element tag and contain information that describes the element content. If you look at markup for the tag `name` (`<name type="nonnumeric" kind="ANS" length="128" offset="4" id="Q244" />`), you are able to observe several attributes associated with this element. An attribute has the form *`name="value"`*. For example, the `type` attribute for the `name` element has a value of `"nonnumeric"`. This information tells the **xmlif** library to obtain data from the COBOL data structure and convert the data from COBOL data format to a text format for the XML document.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- produced by cobtoxml version n.00.00 for RM/COBOL version n.00 or greater on:
  Thur May 15 10:14:26 2003 -->
<!-- data item "customer-address" in program "GETTING-STARTED" in file
  "C:\xmlexample\getstarted.cob" -->
<root type="nonnumeric" kind="GRP" compiledTimeStamp="2003-05-14T12:34:12"
 cobtoxmlRevision="1.0">
 <customer-address type="nonnumeric" kind="GRP" length="454" offset="4" uid="Q1">
  <name type="nonnumeric" kind="ANS" length="128" offset="4" uid="Q2" />
  <address-1 type="nonnumeric" kind="ANS" length="128" offset="132" uid="Q3" />
  <address-2 type="nonnumeric" kind="ANS" length="128" offset="260" uid="Q4" />
  <address-3 type="nonnumeric" kind="GRP" length="70" offset="388" uid="Q5">
   <city type="nonnumeric" kind="ANS" length="64" offset="388" uid="Q6" />
   <state type="nonnumeric" kind="ANS" length="2" offset="452" uid="Q7" />
   <zip type="numeric" kind="NBU" length="4" offset="454" scale="0"
    precision="5" uid="Q8" />
  </address-3>
 </customer-address>
</root>
```

### Internal XSLT Stylesheet File

The XML document, **getstarted.xsl**, is an internal XSLT stylesheet file. XSLT stylesheet files, such as this, are used to transform an XML document into some other data representation (usually, but not necessarily, another XML document). **getstarted.xsl** is used by the **xmlif** library when importing an XML document (converting from XML to COBOL). This internal XSLT stylesheet transforms the imported XML into a new, internal XML document that contains the attributes shown in the template file. This allows the **xmlif** library to convert the text in an XML document to an internal COBOL format and store the data in the appropriate location in the COBOL program's memory.

This internal XSLT stylesheet is complex and performs many additional functions. It is not shown here since it is meaningful only to an experienced XML designer adept at reading and writing XSLT stylesheets.

In addition to internal XSLT stylesheets, the user may code and provide external XSLT stylesheets for importing, exporting, or transforming XML documents.

### Schema File

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

The XML document, **getstarted.xsd**, is a schema file used to validate the content of an XML document. A schema file is a description of how data is structured. Schema files are about the data rather than the data itself. In XML, the term "valid" means that a particular XML document is both well-formed (that is, it has correct XML syntax), and that it is structured and contains content consistent with the constraints intended by the designer of the document. In this case, the **getstarted.xsd** file provides a schema file that would catch errors, such as the entry of a nonnumeric value for a zip code.

There are cases where validation by schema files is not appropriate. In such instances, the **cobtoxml** utility has an option to disable the generation of a schema file, as described in Schema Options (on page 60). Furthermore, the **xmlif** library has options to validate or not validate the content of an XML document, as detailed in XML VALIDATE FILE (on page 71) and XML VALIDATE TEXT (on page 72).

The schema file for the Windows implementation is not presented here because it, too, is meaningful only to an experienced XML designer adept at reading and writing schema files.

**Note** If the application wishes to use several COBOL data structures as separate XML documents within the same COBOL application, it is necessary to run the **cobtoxml** utility once for each data structure, using an optional parameter to provide a name for the model files.

## Execute the COBOL Program

In the fourth step, you execute and test the program.

The following sections explain how—in several stages—you can build upon the preceding steps by adding increasingly more functionality to the COBOL data structure (designed in step 1 of this example), and then compiling and running the program after each stage.

In the first stage, the original program fragment is developed into a working COBOL program that calls the **xmlif** library. Next, the XML EXPORT FILE statement is used to create an XML document from the content of the COBOL data structure. Finally, the XML document is fully populated with data values. With each iteration, the program is recompiled and the **cobtoxml** utility is executed in order to produce the necessary model files.

## *Making a Program Skeleton*

Step 1 started with just a fragment of the program in order to show the COBOL data structure and allow program compilation so that it would be possible to examine the model files generated by the **cobtoxml** utility.

The interface to the **xmlif** library, a COBOL-callable subprogram, is simplified by using some COBOL copy files that perform source text replacement. This means that the developer may write XML commands, which are much like COBOL statements, rather than writing CALL statements that directly access entry points in the **xmlif** library. The COBOL copy files also define program variables that are used in conjunction with the XML commands. The copy file, **lixmlall.cpy**, must be copied in the Working-Storage Section of the program in order to use XML Extensions. For more information, see Copy Files (on page 46).

To call the **xmlif** library, add the following lines (shown in blue) to the COBOL program fragment from step 1:

```
Identification Division.
Program-Id.  Getting-Started.
Data Division.
Working-Storage Section.
01  Customer-Address.
    02  Name          Pic X(128).
    02  Address-1     Pic X(128).
    02  Address-2     Pic X(128).
    02  Address-3.
        03  City      Pic X(64).
        03  State     Pic X(2).
        03  Zip       Pic 9(5) Binary.
Copy "lixmlall.cpy".
Procedure Division.
A.
    XML INITIALIZE.
    If Not XML-OK Go To Z.

< insert COBOL PROCEDURE DIVISION logic here >

Z.
Copy "lixmltrm.cpy".
    GoBack.
Copy "lixmldsp.cpy".
End Program  Getting-Started.
```

The COPY statement is placed in the Working-Storage Section after the COBOL data structure.

The Procedure Division header is entered, followed by the paragraph-name, `A.`.

The XML INITIALIZE statement produces a call to the **xmlif** library. The XML INITIALIZE statement may be thought of as similar to a COBOL OPEN statement.

Termination logic is placed at the end of the program. The paragraph-name, `Z.`, is used as a GO TO target for error or other termination conditions.

The copy file, **lixmltrm.cpy**, is used to generate a correct termination sequence. A call to XML TERMINATE (similar to a COBOL CLOSE statement) is in this copy file. If errors are present, the logic in this copy file will perform a procedure defined in the copy file, **lixmldsp.cpy**, which will display any error messages.

The original program fragment is now a working COBOL program that calls the **xmlif** library. Its only function is to open and close the interface to the library.

**Note** Whenever you recompile the source program, you must run the **cobtoxml** utility again, even if the data structure has not changed. This is necessary because the **xmlif** library must have access to the model files that correctly describe the COBOL data structures. In order to assure this, the **xmlif** library ascertains that the model files were produced from the same object that is being run.

Compile and run the program from the command line as follows:

```
rmcobol  getstarted y
cobtoxml getstarted customer-address
runcobol getstarted
```

The first parameter is the name of the COBOL object program.

If you place the **xmlif** library in the **rmautold** directory, as this action assumes, you do not have to specify the library name on the command line.

## *Making a Program that Exports an XML Document*

The next stage is to create an XML document from the content of a COBOL data structure. To do this, more logic is added to the original COBOL program. The added text is shown in blue.

```
Identification Division.
Program-Id.  Getting-Started.
Data Division.
Working-Storage Section.
01  Customer-Address.
    02  Name          Pic X(128).
    02  Address-1     Pic X(128).
    02  Address-2     Pic X(128).
    02  Address-3.
        03  City      Pic X(64).
        03  State     Pic X(2).
        03  Zip       Pic 9(5) Value 0 Binary.
Copy "lixmlall.cpy".
Procedure Division.
A.
    XML INITIALIZE.
    If Not XML-OK Go To Z.

     XML EXPORT FILE
         Customer-Address
         "Address"
         "getstarted".
     If Not XML-OK Go To Z.

Z.
Copy "lixmltrm.cpy".
    GoBack.
Copy "lixmldsp.cpy".
End Program  Getting-Started.
```

The XML EXPORT FILE statement is used to create an XML document from the content of a COBOL data structure. This statement has three arguments: the data structure name, the desired filename, and the root name of the model files.

A value of zero is added to the zip code field so that the field has a valid numeric value.

As you would expect, the data structure name is `customer-address`. This name must correspond to the name used when running the **cobtoxml** utility (`cobtoxml getstarted customer-address address`). The desired filename is specified as **address**, which will cause a file (containing the XML document) with the name of **address.xml** to be generated. Almost all of the XML statements may set an unsuccessful or warning status value; that is, a status value for which the condition-name `XML-OK` is false following the execution of the XML statement. It is good practice to follow every XML statement with a status test, such as, `If Not XML-OK Go To Z`.

The program is again compiled and run from the command line as follows:

```
rmcobol  getstarted y
cobtoxml getstarted customer-address address
runcobol getstarted
```

This time the program creates an XML document in the example file, **address.xml**. You may use Microsoft Internet Explorer to examine the document. The resulting XML document is displayed as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
 <customer-address>
 <name />
  <address-1 />
  <address-2 />
  <address-3>
   <city />
   <state />
   <zip>0</zip>
  </address-3>
 </customer-address>
</root>
```

Since the data structure contained only spaces (with the exception of the zip field), the generated document is almost identical to the example file that was generated by the **cobtoxml** utility.

## *Populating the XML Document with Data Values*

The next stage is to populate the COBOL program with data values. Changes to the program are again shown in blue.

```
Identification Division.
Program-Id.  Getting-Started.
Data Division.
Working-Storage Section.
01  Customer-Address.
    02  Name          Pic X(128).
    02  Address-1     Pic X(128).
    02  Address-2     Pic X(128).
    02  Address-3.
        03  City      Pic X(64).
        03  State     Pic X(2).
        03  Zip       Pic 9(5) Value 0 Binary.
Copy "lixmlall.cpy".
Procedure Division.
A.
    XML INITIALIZE.
    If Not XML-OK Go To Z.

    Move "Liant Software Corporation" to Name.
    Move "8911 Capitol of Texas Highway, North"
        to Address-1.
    Move "Suite 4300" to Address-2.
    Move "Austin" to City.
    Move "TX" to State.
    Move 78759 to Zip.

     XML EXPORT FILE
         Customer-Address
         "Address"
         "getstarted".
     If Not XML-OK Go To Z.

Z.
Copy "lixmltrm.cpy".
    GoBack.
Copy "lixmldsp.cpy".
End Program  Getting-Started.
```

A series of simple MOVE statements are used to provide content for the data structure.

Again, the program is compiled and run from the command line as follows:

```
rmcobol  getstarted y
cobtoxml getstarted customer-address
runcobol getstarted
```

This time the XML document is fully populated with data values, as shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
 <customer-address>
  <name>Liant Software Corporation</name>
  <address-1>8911 Capitol of Texas Highway North</address-1>
  <address-2>Suite 4300</address-2>
  <address-3>
   <city>Austin</city>
   <state>TX</state>
   <zip>78759</zip>
  </address-3>
 </customer-address>
</root>
```

## Deploy the Application

The final step is to deploy the application. Use the RM/COBOL Combine Program utility, **rmpgmcom**, which is included with the RM/COBOL development system, to strip symbol table information from the COBOL object program. The **rmpgmcom** utility combines multiple RM/COBOL object files into a single program file library. This utility is used primarily to reduce the size of the deployable application.

The following DOS commands illustrate how the **rmpgmcom** utility may be used to strip symbol table information:

```
move /y myprogram.cob tmp.cob

start /w runcobol rmpgmcom A='STRIP,myprogram.cob,tmp.cob'

del tmp.cob
```

Deploy the **xmlif** library and the underlying XML parser that it uses along with the model files that were generated by the **cobtoxml** utility. Normally, these files are stored in the same location as the COBOL program files.

For deploying COBOL applications that use XML Extensions, install the XML Extensions deployment system on each platform that runs the application. You may do this by using the XML Extensions installation disk.

# How XML Extensions Locates Files

Like other RM/COBOL products, XML Extensions uses the following environment variables to locate various files:

- **PATH.** The PATH environment variable is used to locate executable programs, such as **cobtoxml**. This environment variable should contain a reference to the RM/COBOL installation directory, which allows the operating system to locate the **cobtoxml** utility. For example:

  On Windows

  ```
  set PATH=C:\RMCOBOL
  ```

  On UNIX

  ```
  setenv PATH /usr/bin
  ```

- **RMPATH.** The RMPATH environment variable is used by the RM/COBOL compiler to locate source files. This environment variable should contain a reference to the RM/COBOL installation directory, which allows the RM/COBOL compiler to locate copy files that are referenced by COBOL programs that use XML statements. For example:

  On Windows

  ```
  set RMPATH=C:\RMCOBOL
  ```

  On UNIX

  ```
  setenv RMPATH=/usr/rmcobol
  ```

- **RUNPATH.** The RUNPATH environment variable is used by the RM/COBOL runtime and by the **xmlif** support module (a 32-bit dynamic link library on Windows named **xmlif.dll**, and a shared object on UNIX named **xmlif.so**) to locate files at runtime. For example:

  On Windows

  ```
  set RUNPATH=C:\MYFILES
  ```

  On UNIX

  ```
  setenv RMPATH=/usr/myfiles
  ```

  The use of RUNPATH by the **xmlif** support module is similar but not completely identical to that used by the RM/COBOL runtime. The RUNPATH search sequence for XML Extensions has been modified to ignore directory names that use the Universal Naming Convention (UNC) notation (for example, "//system/directory"). UNC names are normally used in an application that uses RM/InfoExpress. XML Extensions cannot access files directly through RM/InfoExpress. By ignoring UNC directory names, unnecessary time delays are avoided when performing a RUNPATH search.

For additional information locating files, see the following topics:

- Automatic Search for Files (on page 37)
- File Naming Conventions (on page 38)
- UNIX Character Encoding (on page 41)
- Windows Character Encoding (on page 41)

# Chapter 3: COBOL Considerations

This chapter provides information specific to using RM/COBOL when developing an XML-enabled application. The primary topics discussed in this chapter include the following:



## File Management

The management of data files when using XML Extensions for RM/COBOL is similar, but not identical, to other RM/COBOL data file management issues. These issues include the following:



### Automatic Search for Files

During development with XML Extensions, remember the following points when searching for a file not found in the current working directory:

- The RM/COBOL runtime support for resolving leading or subsequent names in a path name is not provided by XML Extensions when the **xmlif** library locates files. That is, XML Extensions does not honor the RESOLVE-LEADING-NAME or RESOLVE-SUBSEQUENT-NAMES keywords of the RUN-FILES-ATTR configuration record.
- The RUNPATH environment variable is searched to locate the XML model files, as necessary. XML model files, such as template files (***modelname*.xtl**)

and internal XSLT stylesheet files (***modelname*.xsl**), are produced when the **cobtoxml** utility processes the symbol table of a previously compiled RM/COBOL object file.

- If the RUNPATH environment variable contains UNC references (directory names beginning with "//" or "\\"), the **xmlif** library will skip those names. UNC references typically refer to foreign file systems that are accessed through RM/InfoExpress. These names are skipped in order to avoid server performance degradation.
- The RUNPATH environment variable is also searched to locate input XML data document files and all external XSLT stylesheet files.

For more information, see Model Files (on page 28) and Referencing XML Model Files (on page 61).

## File Naming Conventions

File extensions are either used "as is" or forced to be a predetermined value. The conventions governing particular filename extensions when using XML Extensions are described in the topics that follow.

**Note** A filename extension is never added if the filename is a URL, that is, begins with "http:" or "https:".

### *Model File Naming Conventions*

Model files, the XML documents generated by the **cobtoxml** utility, have predetermined extensions. The **cobtoxml** utility generates a set of three (or four) files from a single filename with different extensions. A set of model files consist of the following:

- One example file (**.xml**)
- One template file (**.xtl**)
- One internal XSLT stylesheet file (**.xsl**)
- One schema file (**.xsd**)

  **Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

For a more detailed discussion, see Model Files (on page 28).

The **xmlif** library uses the model files only as input files. When the **xmlif** library references a model file, the appropriate predetermined extension is added, regardless of the presence or lack of an extension on the model file parameter supplied by the COBOL program. For more information, see Referencing XML Model Files (on page 61).

The **xmlif** library uses the RUNPATH environment variable to locate a model file (with the appropriate extension added) *except* when:

- the model filename contains a directory separator character (such as "\" on Windows);
- the file exists; or
- the filename is a URL (that is, the name begins with "http:" or "https:").

### External XSLT Stylesheet File Naming Conventions

In addition to the internal XSLT stylesheet that is produced as part of the model file set, external XSLT stylesheets may be referenced by the **xmlif** library. If the filename parameter supplied by the COBOL program does not contain an extension, the value **.xsl** is added to the filename.

The **xmlif** library uses the RUNPATH environment variable to locate an external XSLT stylesheet file (with the **.xsl** extension added) *except* when:

- the external XSLT stylesheet filename parameter supplied by the COBOL program contains a directory separator character (such as "\" on Windows);
- the file exists; or
- the filename is a URL (the name begins with "http:" or "https:").

### Other Input File Naming Conventions

All other input files referenced by the **xmlif** library will have a value of **.xml** added if the filename parameter supplied by the COBOL program does not contain an extension. No RUNPATH environment variable search is applied.

### Other Output File Naming Conventions

All other output files referenced by the **xmlif** library will have a value of **.xml** added if the filename parameter supplied by the COBOL program does not contain an extension. No RUNPATH environment variable search is applied.

If the filename supplied by the COBOL program is a URL, then an error is returned because it is not possible to write directly to a URL.

## Data Conventions

In XML Extensions for RM/COBOL, several suppositions have been made about data transformations between COBOL and XML, including those relating to the following issues:

- Data representation (on page 40)
- FILLER data items (on page 42)
- Missing intermediate parent names (on page 43)
- Sparse COBOL records (on page 45)

# Data Representation

COBOL numeric data items are represented in XML as a numeric string. A leading minus sign is added for negative values. Leading zeros (those appearing to the left of the decimal point) are removed. Trailing zeros (those appearing to the right of the decimal point) are likewise removed. If the value is an integer, no decimal point is present.

COBOL nonnumeric data items are represented as a text string and have trailing spaces removed (or leading spaces, if the item is described with the JUSTIFIED phrase). In addition, any embedded XML special characters are represented by escape sequences; the ampersand (&), less than (<), greater than (>), quote (”), and apostrophe (‘) characters are examples of such XML special characters.

On Windows platforms, nonnumeric displayable data are normally encoded using Microsoft’s OEM data format. On output, these data are converted to the standard Unicode 8-bit transformation format, UTF-8. On input, data is converted to the OEM data format. If the XML SET ENCODING (on page 85) statement is used to specify “UTF-8”, then the internal data format is UTF-8. For more information, see the discussion of Windows character encoding on page 41.

On UNIX platforms, nonnumeric displayable data are normally encoded using a “local” character encoding that the UNIX system uses. Typically, this may be Latin-1 or Latin-9. On output, these data are converted to the standard Unicode 8-bit transformation format, UTF-8. On input, data is converted to the systems internal format. If the XML SET ENCODING statement is used to specify “UTF-8”, then the internal data format is UTF-8. For more information on selecting an appropriate “local” character encoding, refer to the discussion of UNIX character encoding on page 41.

## *COBOL and Character Encoding*

The **xmlif** library uses UTF-8 character encoding for exporting XML documents. (UTF-8 is a byte-oriented encoding form of Unicode that has been designed for ease-of-use with existing ASCII-based systems.) Imported documents are interpreted according to the character encoding specified in the XML header, resulting in an internal Unicode representation of the characters. Because XML is Unicode-based and RM/COBOL is not, a transcoding is generally required when moving character data between COBOL and XML. The **xmlif** library supports various means of specifying the transcoding that should occur in these cases. The following sections have related information regarding character encoding considerations.

### RM_ENCODING Environment Variable

The RM_ENCODING environment variable is used to specify the “local” character encoding. This environment variable is ignored if the XML SET ENCODING (on page 85) statement sets the encoding to UTF-8. The interpretation of this environment variable also varies between Windows and UNIX character encoding, as discussed in the next topics.

## Windows Character Encoding

Under Windows, the RM/COBOL runtime uses OEM character encoding. Therefore, the Windows implementation of XML Extensions also supports OEM character encoding. The RM_ENCODING environment variable is ignored by the Windows implementation of XML Extensions.

**Note** Microsoft originally introduced OEM character encoding for MS-DOS. While there are multiple OEM code pages in use, the Windows operating system provides interfaces that allow conversion between the OEM code page in use and Unicode. XML Extensions does not need to differentiate between code pages.

## UNIX Character Encoding

Under UNIX, the RM/COBOL runtime is normally not concerned with the data encoding used by the underlying operating system. Liant, however, has decided that Latin-1 (ISO-8859-1) is important for the U.S. and that Latin-9 (ISO-8859-15) is significant for Western Europe because it contains the Euro currency symbol.

The RM_ENCODING environment variable (on page 40) may specify the built-in and predefined values of RM_LATIN_1 and RM_LATIN_9. These values are used to designate that either Latin-1 or Latin-9 is being used as the local character encoding. Internal translation functions convert between either Latin-1 or Latin-9 (in COBOL memory) and UTF-8 (in the XML document). The value of the environment variable is case insensitive with hyphen and underscore characters being optional. For example, “RM_LATIN_9”, “Rm-Latin-9”, and “rmlatin9” are equivalent.

If the value of the RM_ENCODING environment variable is not specified, then RM_LATIN_9 is used as the default.

If the value of the RM_ENCODING environment variable is specified with a value that is not RM_LATIN_1 or RM_LATIN_9, then the value that is passed must be a name recognized by the **iconv** library. The **iconv** library can perform other conversions. In this case, the spelling may need to be exact (for example, the value may be case sensitive, and hyphens and underscores would be required). The exact spelling of the value of the RM_ENCODING environment variable is specific to the **iconv** library on the platform in use.

**Note** Liant does not provide an **iconv** library. The developer must acquire an appropriate package.

The value of the RM_ICONV_NAME environment variable, if one is defined, is used to locate the **iconv** library (which must be a shared object) on the local system. For example:

```
RM_ICONV_NAME=/usr/local/bin/libiconv.so
```

If the RM_ICONV_NAME environment variable is not set, then the PATH environment variable is searched for either of the specific names, **iconv.so** or **libiconv.so** (in that order).

## FILLER Data Items

Unnamed data description entries, referred to as FILLER data items in this section, may be used to generate XML text without starting a new XML element name. Specifying named and unnamed elementary data items subordinate to a named group generates XML mixed content for an element named by the group name.

Numeric FILLER data items will not reliably produce well-formed XML sequences. For this reason, FILLER data items should always be nonnumeric PIC X or PIC A.

For example, the following COBOL sequence:

```
01  A.
    02  FILLER  Value "ABC".
    02  B       Pic X(5) Value "DEF".
    02  FILLER  Value "GHI".
```

generates the following well-formed XML sequence:

```
<a>ABC<b>DEF</b>GHI</a>
```

FILLER data items, however, are treated differently than named data. All leading and/or trailing spaces are preserved, so that the length of the data is the same as the COBOL data length.

In addition, the data is treated as PCDATA. That is, embedded XML special characters are preserved. This allows short XHTML sequences, such as “break” to be represented as FILLER (for example, <br />). XHTML (eXtensible HyperText Markup Language) is based on HTML 4, but with restrictions such that an XHTML document is also a well-formed XML document. For example, the following COBOL sequence:

```
01  A.
    02  FILLER  Value "<br />".
    02  B       Pic X(5) Value "DEF".
    02  FILLER  Value "GHI".
```

generates the following well-formed XML sequence:

```
<a><br /><b>DEF</b>GHI</a>
```

Care must be taken in placing XML special characters in FILLER data items, since the resultant XML sequence might not be well-formed. For example, the following COBOL sequence:

```
01  A.
    02  FILLER  Value "<br".
    02  B       Pic X(5) Value "DEF".
    02  FILLER  Value "GHI".
```

generates the following syntactically malformed XML sequence:

```
<a><br<b>DEF</b>GHI</a>
```

Whenever FILLER data items are present in a data item that is referenced by the XML EXPORT statements, the resulting document is checked to ensure that the resultant XML document is well-formed.

## Missing Intermediate Parent Names

A capability for handling missing intermediate parent names has been included to make programs that deal with “flattened” data items, such as Web services, less complicated.

Sometimes it is possible for XML Extensions to reconstruct missing intermediate parent names in a COBOL data structure. These missing names may be generated in either of two ways:

- Unique element names (see the next topic). Use this technique to determine whether the element name is unique. If this is true, then the intermediate parent names are generated by the internal XSLT stylesheet model file.
- Unique identifier (on page 44). Use this method to determine whether the unique identifier (uid) attributes of the element name are provided. If this is true, then the intermediate parent names may also be generated.

### *Unique Element Names*

Consider the following COBOL data structure:

```
01  Liant-Address.
    02  Name          Pic X(64).
    02  Address-1     Pic X(64).
    02  Address-2     Pic X(64).
    02  Address-3.
        03  City      Pic X(32).
        03  State     Pic X(2).
        03  Zip       Pic 9(5).
    02  Time-Stamp    Pic 9(8).
```

A well-formed and valid XML document that could be imported into this structure is shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
    <name>Liant Software Corporation</name>
    <address-1>8911 Capital of Texas Highway North</address-1>
    <address-2>Suite 4300</address-2>
    <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78759</zip>
    </address-3>
    <time-stamp>13263347</time-stamp>
  </liant-address>
</root>
```

A well-formed (but not valid) “flattened” version of an XML document that could also be imported into this structure is displayed here:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <name>Wild Hair Corporation</name>
  <address-1>8911 Hair Court</address-1>
  <address-2>Sweet 4300</address-2>
  <city>Lostin</city>
  <state>TX</state>
  <zip>70707</zip>
  <time-stamp>99999999</time-stamp>
</root>
```

**Note** This last XML document may be used only if the **cobtoxml** utility does not generate a schema to validate the document. To prevent the creation of a schema file, use the **-sn** (schema none) option on the **cobtoxml** utility. You may also delete an existing schema model file (**.xsd**) or choose not to deploy the schema model file with the application. Furthermore, under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

### *Unique Identifier*

The unique identifier (uid) attribute is generated by an XML EXPORT FILE (on page 64) or XML EXPORT TEXT (on page 66) statement if XML attributes are enabled. Attributes may be enabled by using the XML ENABLE ATTRIBUTES (on page 82) statement before the XML EXPORT statements.

Using the same COBOL data structure illustrated for unique element names (described in the previous section), a well-formed XML document (generated by XML EXPORT), which contains attributes—including uids, that could be imported into this structure is shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root type="nonnumeric" kind="GRP"
 compiledTimeStamp="2003-05-14T10:57:22" cobtoxmlRevision="1.0">
  <liant-address type="nonnumeric" kind="GRP" length="239" offset="4"
   uid="Q1">
    <name type="nonnumeric" kind="ANS" length="64" offset="4"
     uid="Q2">Liant Software Corporation</name>
    <address-1 type="nonnumeric" kind="ANS" length="64" offset="68"
     uid="Q3">8911 Capital of Texas Highway North</address-1>
    <address-2 type="nonnumeric" kind="ANS" length="64" offset="132"
     uid="Q4">Suite 4300</address-2>
    <address-3 type="nonnumeric" kind="GRP" length="39" offset="196"
     uid="Q5">
      <city type="nonnumeric" kind="ANS" length="32" offset="196"
       uid="Q6">Austin</city>
      <state type="nonnumeric" kind="ANS" length="2" offset="228"
       uid="Q7">TX</state>
      <zip type="numeric" kind="NSU" length="5" offset="230" scale="0"
       precision="5" uid="Q8">78759</zip>
    </address-3>
    <time-stamp type="numeric" kind="NSU" length="8" offset="235"
     scale="0" precision="8" uid="Q9">10572765</time-stamp>
  </liant-address>
</root>
```

A well-formed “flattened” version of an XML document that could also be imported into this structure is displayed below. The uid attributes were captured from an XML document (such as the one shown previously) that was generated by an XML

EXPORT statement. These attributes may be captured by an XSLT stylesheet or other process, and then added again before the XML IMPORT FILE (on page 67) or XML IMPORT TEXT (on page 68) statement. This is accomplished by combining the element name and the uid attribute value to form a new element name. For example, <name uid="Q2">, could be used to generate a new element name "name.Q2".

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <name uid="Q2">>Wild Hair Corporation</name>
  <address-1 uid="Q3">>8911 Hair Court</address-1>
  <address-2 uid="Q4">>Sweet 4300</address-2>
  <city uid="Q6">Lostin</city>
  <state uid="Q7">TX</state>
  <zip uid="Q8">70707</zip>
  <time-stamp uid="Q9">99999999</time-stamp>
</root>
```

**Note** This last XML document may be used only if the **cobtoxml** utility does not generate a schema to validate the document. To prevent the creation of a schema file, use the **-sn** (schema none) option on the **cobtoxml** utility. You may also delete an existing schema model file (**.xsd**) or choose not to deploy the schema model file with the application. Furthermore, under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

## Sparse COBOL Records

An input XML document need not contain all data items defined in the original structure. This applies to both scalar and array elements. In order to place array elements correctly, a subscript must be supplied when array elements are not in canonical order.

For example, the following XML document uses the subscript attribute to position the array to the second element and then to the fourth element.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <data-table>
    [
    <table-1 subscript="2">
      <x>B</x>
      <n>2</n>
    </table-1>
    <table-1 subscript="4">
      <x>D</x>
      <n>4</n>
    </table-1>
    ]
  </data-table>
</root>
```

If the input XML document might be sparse (that is, missing some elements), then the schema generated by the **cobtoxml** utility will cause the document load to fail. For this reason, if you anticipate using sparse XML documents, you should run the **cobtoxml** utility with the **-sn** (schema none) option. You may also delete an existing schema model file (**.xsd**) or choose not to deploy the schema model file with the application.

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

# Copy Files

Under most circumstances, you should make use of the copy files that are provided in XML Extensions for RM/COBOL. Various points to consider when using copy files with XML Extensions include the following:

- Statement definitions (as discussed in the following topic)
- Displaying status information (as discussed below)
- Application termination (on page 47)

## Statement Definitions

The copy file, **lixmlall.cpy**, is required to define the XML statements and to define some data-items that are referenced. This copy file should be copied in the Working-Storage Section of the program. In general, do not modify the contents of this copy file or the copy files that it copies (**lixmdef.cpy** and **lixmlrpl.cpy**). An exception to this practice is discussed in the following topic.

## Displaying Status Information

The copy file, **lixmldsp.cpy**, is provided as an aid in retrieving and presenting status information. This copy file defines the Display-Status paragraph and contains the following text:

```
Display-Status.
    If Not XML-IsSuccess
        Perform With Test After Until XML-NoMore
            XML GET STATUS-TEXT
            Display XML-StatusText
        End-Perform
    End-If.
```

The DISPLAY statement, `Display XML-StatusText`, displays status information on the terminal display. You may edit this statement, as necessary, for your application. For example, the definition of the `XML-StatusText` field in the **lixmldef.cpy** copy file may be altered from the default of 80 to change the size of the buffer used to contain XML status information.

While this logic is normally used in the application termination logic, it may be used at any time in the program flow. For example:

```
    XML TRANSFORM FILE "A" "B" "C".
    Perform Display-Status.
```

## Application Termination

The copy file, **lixmltrm.cpy**, provides an orderly way to shut down an application. This copy file contains the following text:

```
    Display "Status: " XML-Status.
    Perform Display-Status.
    XML TERMINATE.
    Perform Display-Status.
```

The first line may be modified or removed, as you choose. The first PERFORM statement displays any pending status messages (from a previous XML statement). The XML TERMINATE statement shuts down XML Extensions. The second PERFORM statement displays any status from the XML TERMINATE operation.

The following logic is sufficient to successfully terminate XML Extensions:

```
Z.
Copy "lixmltrm.cpy".
    Stop Run.
Copy "lixmldsp.cpy".
```

The `Z.` paragraph-name is where the exit logic begins. The flow of execution may reach here by falling through from the previous paragraph or as the result of a program branch. The STOP RUN statement is used to prevent the application from falling through to the Display-Status paragraph. An EXIT PROGRAM or GOBACK statement also may be used, if appropriate.

# Miscellaneous Considerations

This section describes a number of items related to using XML Extensions with RM/COBOL, including the following:

- Anonymous COBOL data structures (on page 48)
- Relaxed time stamp checking (on page 48)

## Anonymous COBOL Data Structures

XML Extensions now supports the use of an anonymous COBOL data structure when exporting and importing documents. An anonymous data structure is any data area that is the same size or larger than the data structure indicated by the template file. This means that exporting or importing can be done to Linkage Section data items that are based on either arguments passed to a called program or a pointer using the SET statement (for example, into allocated memory). Importing and exporting can also occur with data items having the external attribute. (An *external attribute* is the attribute of a data item obtained by specification of the EXTERNAL clause in the data description entry of the data item or of a data item to which the subject data item is subordinate.)

## Relaxed Time Stamp Checking

Time stamp checking is used to guarantee that the correct model files are referenced in the XML import and export statements. (These statements are described in the topic, Document Processing Statements on page 64). In the first release of XML Extensions, it was necessary for the compilation time stamp in the object program to match the **cobtoxml** time stamp in the template file. However, the program may now be recompiled without running the **cobtoxml** utility. It is necessary to run **cobtoxml** only when the relevant data structure(s) are changed. Therefore, it is the programmer's responsibility to specify the correct and current model file. Specifying an incorrect or non-current model file may result in the wrong data being exported or imported.

# Limitations

This section describes the limitations of XML Extensions for RM/COBOL and the way in which those limitations affect the development of an XML-enabled application. The topics discussed in this context include:



## Data Items (Data Structures)

Data items that are passed to XML Extensions must be in memory that is local to the COBOL program. Therefore, EXTERNAL data items or data items in the Linkage Section may not be used for import or export operations.

The import and export statements operate on a single COBOL data item. This data item is the second command line parameter when using the **cobtoxml** utility. As you would expect, this data item may be (and usually will be) a group item. The COBOL program must move all necessary data to the selected data item before using the

XML EXPORT FILE (on page 64) or XML EXPORT TEXT (on page 66) statements and retrieve data from the data item after using the XML IMPORT FILE (on page 67) or XML IMPORT TEXT (on page 68) statement.

The referenced data item—and any items contained within it, if it is a group item—has the following limitations:

1. REDEFINES and RENAMES clauses are not allowed.
2. FILLER data items must be nonnumeric.
3. The data item must be the same size or larger than the data item specified when building the model files with the **cobtoxml** utility, but it is not required to be the same data item. For additional information, see Anonymous COBOL Data Structures (on page 48).

## Edited Data Items

Numeric edited, alphabetic edited, and alphanumeric edited data items are allowed. The data items are represented in an XML document in the same format as the data items would exist in COBOL internal storage. That is, no editing or de-editing operations are performed for edited data items during import from XML or export to XML. Leading and trailing spaces are preserved.

## Wide and Narrow Characters

XML was developed to use wide (16-bit) Unicode characters as its natural mode. RM/COBOL uses narrow (8-bit) ASCII characters. All XML data that is generated by XML Extensions is represented in UTF-8 format, which is essentially ASCII with extensions for representing 16-bit and larger characters and is compatible with Unicode. (UTF-8 is a form of Unicode.)

## Data Item Size

By its nature, XML has no limits on data item size. COBOL does have size limitations for its data items. Many XML documents have been standardized and such standards include limitations on data items, but the COBOL program must still be written to deal with data item size constraints.

## OCCURS Restrictions

Although, XML has no limits on the number of occurrences of a data item, COBOL does have such occurrence limits. As with data item size, the COBOL program must deal with this difference.

## Reading, Writing, and the Internet

It is possible to read any XML document (including XML model files) from the Internet via a URL. However, it is not possible to write or export an XML document directly to the Internet via a URL.

# Optimizations

Some optimizations have been added to the **xmlif** library to improve performance and reduce the size of the generated documents. Refer also to Chapter 6: *xmlif Library Reference* (on page 63) for more information.

## Occurs Depending

As expected, on output, the XML export statements will limit the number of occurrences of a group to the value of the DEPENDING variable. Additional occurrences may be omitted if they contain no data. For more information, see Empty Occurrences (on page 50).

On input, the XML import statements will store the value of the DEPENDING variable. The XML import statements will also store all occurrences in the document (up to the maximum occurrence limit), regardless of the value of the DEPENDING variable. However, if a schema file is generated by the **cobtoxml** utility, the schema file will report an error if not all of the elements specified by the DEPENDING variable are present.

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

## Empty Occurrences

On output, the XML EXPORT FILE (on page 64) or XML EXPORT TEXT (on page 66) statements recognize occurrences within a group that contain no information. Specifically, an empty data item is a string that contains either all spaces or zero characters, or a number that contains a zero value.

If all of the elementary data items in an occurrence of a group are empty and if the occurrence is not the first occurrence, then no data is generated for that occurrence. This prevents the repetition of occurrences that contain no information.

On input operations with XML IMPORT FILE (on page 67) or XML IMPORT TEXT (on page 68) statements, a schema file may detect an error if not all expected occurrences of an item are present. In order to prevent this, you may enable all occurrences using the XML ENABLE ALL-OCCURRENCES (on page 81) statement, when generating the document (with XML export operations).

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

## Cached XML Documents

Since XSLT stylesheet, template, and schema documents are largely invariant, performance can usually be improved by caching previously loaded versions of these documents in memory.

For some applications, it may be useful to disable caching. If XSLT stylesheet, template, or schema files are generated or replaced in real time, then the cached documents would need to be replaced as well.

If system resource availability becomes critical because a large number of documents are occupying virtual memory, then caching may cause system degradation.

Several XML statements may be used to enable or disable document caching. These statements include:

- XML ENABLE CACHE (on page 83)
- XML DISABLE CACHE (on page 82)
- XML FLUSH CACHE (on page 83)

By default, caching is enabled.

# Chapter 4: XML Considerations

This chapter provides information specific to using XML when using XML Extensions with RM/COBOL to develop an XML-enabled application. The primary topics discussed in this chapter include:

- XML and character encoding (as discussed in the following topic)
- Document type definition support (as discussed below)
- XSLT stylesheet files (on page 55)
- Schema files (on page 55)

## XML and Character Encoding

For internal representation, XML documents use the Unicode character encoding standard. Unicode represents characters as 16-bit items. For external representation, most XML documents are encoded using the standard Unicode transformation formats, UTF-8 or UTF-16. XML documents created by XML Extensions are always encoded for external presentation using the UTF-8 representation. UTF-8 is a method of encoding Unicode where most displayable characters are represented in 8-bits. Characters in the range of 0x20 to 0x7e (the normal displayable character set) are indistinguishable from standard ASCII.

## Document Type Definition Support

It is possible to specify a document type definition (DTD) when exporting XML documents. A DTD can be used to define entity names that are referred to by the values of FILLER data items in the COBOL data structure being exported. For example, if the COBOL data structure is generated using the **webtocobol** utility from a page layout created by Microsoft Front Page, one or more FILLER data items may have entity references for non-breaking spaces ( ). The nbsp entity is not among the predefined XML entities, which are presumed to be declared in all XML documents. (The predefined XML entities are lt, gt, amp, apos, and quot.) If the COBOL data structure containing the nbsp entity reference is exported, it will not be valid XML unless a DTD is provided that causes the nbsp entity to be declared. XML entities can only be declared within a DTD.

The XML EXPORT FILE (on page 64) and XML EXPORT TEXT (on page 66) statements allow specifying a document prefix parameter. The document prefix parameter provides a string, which is exported between the XML header and the first element of the document. This string may specify a DTD when one is needed. The DTD may declare entities in the internal subset directly in the provided DTD, or it may specify an external subset through a URL reference. For example, the nbsp entity is declared in the HTML markup declarations, which can be accessed with a prefix of the form:

```
78 HTML-Entity-Defs          VALUE
   "<!DOCTYPE root [" &
   "<!ENTITY % HTMLlat1 PUBLIC " &
   """-//W3C//ENTITIES Latin 1 for XHTML//EN"" " &
   """http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent""" &
   "> %HTMLlat1; " &
   "<!ENTITY % HTMLsymbol PUBLIC " &
   """-//W3C//ENTITIES Symbols for XHTML//EN"" " &
   """http://www.w3.org/TR/xhtml1/DTD/xhtml-symbol.ent""" &
   "> %HTMLsymbol; " &
   "<!ENTITY % HTMLspecial PUBLIC " &
   """-//W3C//ENTITIES Special for XHTML//EN"" " &
   """http://www.w3.org/TR/xhtml1/DTD/xhtml-special.ent""" &
   "> %HTMLspecial;]>".
```

In contrast, Example C: Export File with Document Prefix (on page 167), demonstrates using an internal subset to declare entities in the DTD itself.

When a document prefix is specified in the XML export statements, XML Extensions will cause the document to be loaded after it is created. This load of the document will verify that the created document is well-formed.

**Note** In the Windows implementation, the Microsoft MSXML parser 4.0 ignores the DTD when validating an XML document against a schema file. Any entities declared in the DTD will not be defined and cannot be referenced. If any entities other than the predefined XML entities are referenced, the document is not well-formed and will fail to load, much less validate. Any XML document that contains entity references, other than the predefined XML entities, must be transformed with an XSLT stylesheet prior to validation against a schema file when using the Microsoft MSXML parser 4.0. For more information, see Schema Files (on page 55).

A DTD is also required in an external XSLT stylesheet that uses entity references other than the predefined XML entity references. Using non-predefined entity references commonly occurs when the XSLT stylesheet is generated by tools for generating transformations from XML to HTML or XHTML, that is, to generate a page to be displayed by a browser. Often the tool will not add the DTD. A DTD that defines the entities must be added after the XSLT stylesheet is generated and before it is used by XML Extensions. Here is an example of such a DTD for XHTML entities:

```
<!DOCTYPE root [
   <!ENTITY % HTMLlat1 PUBLIC
     "-//W3C//ENTITIES Latin 1 for XHTML//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent"> %HTMLlat1;
   <!ENTITY % HTMLsymbol PUBLIC
       "-//W3C//ENTITIES Symbols for XHTML//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml-symbol.ent"> %HTMLsymbol;
   <!ENTITY % HTMLspecial PUBLIC
      "-//W3C//ENTITIES Special for XHTML//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml-special.ent"> %HTMLspecial;]>
```

# XSLT Stylesheet Files

XSLT stylesheet files are used to transform an XML document into another XML document or another type of document—not necessarily in XML format; for example, HTML, PDF, RTF, and so forth. An XSLT stylesheet is an XML document. The **xmlif** library has a specific statement, XML TRANSFORM FILE (on page 70), which is used for performing XSLT stylesheet transformations. In addition, the import and export statements, XML IMPORT FILE (on page 67), XML IMPORT TEXT (on page 68), XML EXPORT FILE (on page 64), and XML EXPORT TEXT (on page 66), allow an external XSLT stylesheet to be specified as a parameter, making it possible to transform a document while importing or exporting XML documents. One of the model files generated by the **cobtoxml** utility is the internal XSLT stylesheet, which is applied automatically during import of an XML document into COBOL. The internal XSLT stylesheet is applied after the external XSLT stylesheet where the XML IMPORT FILE/TEXT statement specifies the optional external XSLT stylesheet parameter.

The format of XML documents generated by XML Extensions matches the form of the specified COBOL data structure. Often the COBOL developer must process XML documents that are defined by an external source. It is likely that the format of the COBOL-generated XML document will not conform to the document format that meets the external requirements.

The recommended course of action is to use an external XSLT stylesheet file to transform between the COBOL-generated XML document format and the expected document format. XSLT stylesheets are extremely powerful. You may wish to use an XSLT stylesheet editing tool to design your XSLT stylesheets (for example, Microsoft's BizTalk Mapper, which is part of BizTalk Server 2000).

Keep in mind that XSLT stylesheets are unidirectional. Therefore, it is possible that you will have to design two external XSLT stylesheets for each COBOL data structure: one for input, which converts the required document format to COBOL format, and one for output, which converts COBOL format to the required external format.

# Schema Files

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

Schema files are used to assure that the data within an XML document conforms to expected values. For example, an element that contains a zip code may be restricted to a numeric integer. Schema files can also limit the length or number of occurrences of an element as well as guarantee that elements occur in the expected order.

A schema file may be applied to an XML document using any of the following methods:

- The entire schema file may reside within the document. (This situation is rare.)
- A link to the schema file may be placed in the document. (This technique is more common.)
- A process that loads a given XML document may also load a schema file that controls the document.

It is this third approach that is used by the **xmlif** library in XML Extensions for RM/COBOL. The **cobtoxml** utility optionally generates a schema file as one of the model files. This schema file is used to validate XML documents that are loaded by the XML IMPORT FILE or XML IMPORT TEXT statements. To skip this validation, specify the **-sn** (schema none) option on the **cobtoxml** utility to disable the generation of a schema file or simply delete the schema file (**.xsd**).

**Note** In the Windows implementation, the Microsoft MSXML parser 4.0 ignores the document type definition (DTD) when validating an XML document against a schema file. Any entities declared in the DTD will not be defined and cannot be referenced. If any entities other than the predefined XML entities are referenced, the document is not well-formed and will fail to load, much less validate. Thus, when a DTD is generated to define entities in an exported document, either the **cobtoxml** option to suppress generation of the schema file should be specified if the exported document is also imported "as is" or the exported document should be transformed prior to being imported so as not to contain entity references. See Document Type Definition Support (on page 53) for information on generating a DTD to define entity references.

# Chapter 5: cobtoxml Utility Reference

This chapter describes the **cobtoxml** utility used by XML Extensions for RM/COBOL and the XML document files, known as model files, that are produced when the **cobtoxml** utility processes the symbol table of a previously compiled RM/COBOL object file.

## What is the cobtoxml Utility?

The **cobtoxml** utility is an application program. It processes the symbol table of a previously compiled RM/COBOL object file and produces a set of XML documents. These XML documents are called model files and are described in more detail in Model Files (on page 28) and Referencing XML Model Files (on page 61).

To use the **cobtoxml** utility, you specify (at a minimum) the name of a COBOL object file and the name of a COBOL data item within that file. You may use the **cobtoxml** utility multiple times against the same object file to process different data items.

The **cobtoxml** utility requires that the COBOL object program be compiled with the RM/COBOL Compile Command Y Option enabled in order to place symbol table information in the object file. However, since there are no runtime requirements for the symbol table, the symbol table may be removed once applications are ready to be deployed.

**Note** An older version of the **cobtoxml** utility may be unable to handle an object produced by a newer RM/COBOL compiler. This can occur because the COBOL object version that is produced by the newer compiler contains features that are unsupported and unavailable in the earlier release of XML Extensions. In such cases, the RM/COBOL Compile Command Z Option may allow you to force the compiler to produce an object acceptable to **cobtoxml**.

# Command Line Interface

The **cobtoxml** utility (**cobtoxml.exe** on Windows and **cobtoxml** on UNIX) is executed with the following command:

```
cobtoxml cob-file-name data-item-name [model-file-name] [options]
```

*cob-file-name*, the first positional parameter, is the name of the RM/COBOL object file. The RM/COBOL source program must have been compiled with the symbol table option specified by the RM/COBOL Compile Command Y Option. The value of this name is treated as case sensitive. If this parameter contains an extension, it will be used as entered. If the extension is omitted, **.cob** will be added. No directory search (on the PATH or RMPATH environment variables) is performed.

*data-item-name*, the second positional parameter, is the name of the selected data item within a COBOL program. While the most common use may be as the name of a group, the data item need not be a record name (01 level). The value entered for this parameter is not case sensitive. The data-name must be defined exactly once in the program file. In the case of program libraries, all programs are searched.

*model-file-name*, the optional third parameter, is the base name used to create a set of XML documents, called model files, having a single filename with different, predetermined extensions (**.xml**, **.xtl**, .**xsl**, and **.xsd**). The value of this name is treated as case sensitive. If this parameter already contains an extension, it will be ignored.

*options* represents command line options, which are described in Command Line Options (on page 59). Although this parameter is shown as the last parameter, it may occur anywhere after **cobtoxml** on the command line. Additionally, *options* may be specified multiple times. Option letters are case insensitive; that is, the following combinations are equivalent: “-bc”, “-bC”, “-Bc” and “-BC”. The *options* parameter is divided into three categories: banner, name, and schema.

**Note** Under Windows, when no command line parameters are entered, the following **cobtoxml.exe** usage message is displayed:

```
RM/COBOL cobtoxml utility - Version nn.nn.nn for 32-Bit Windows.
Copyright (c) 2001-2nnn by Liant Software Corp.  All rights reserved.

Usage:  cobtoxml cob-file-name data-item-name model-file-name
  cob-file-name: case-sensitive name of the RM/COBOL object file
  data-item-name: case-insensitive name of the COBOL data item
  model-file-name: optional case-sensitive name for the XML file(s)
  options: a sequence of option letters preceded by a hyphen
```

# Command Line Options

The following options are available on the **cobtoxml** command line: banner, name, and schema.

## *Banner Options*

The banner options control the amount of information displayed during the execution of the **cobtoxml** utility. A banner option is created by entering a hyphen character (-) followed by the letter "b" and then by one of the following letters: "c", "n", or "v".

The following table lists several examples of supported banner option combinations:

| Option | Description |
|---|---|
| -bc | Displays the Liant copyright message only. (This is the default.) |
| -bn | Displays no banner information. |
| -bv | Displays verbose banner. |

Banner options do not affect the display of any error or status messages.

## *Name Options*

The name options control the format of tag names in XML documents. An XML tag is generated for each data-name in the specified COBOL data structure. Since COBOL data-names are case insensitive and XML tag names are case sensitive, it is necessary to have rules for generating XML tag names. By default, the **cobtoxml** utility generates XML tag names in lowercase.

A name option is generated by entering a hyphen character (-) followed by the letter "n" and then by one or more of the following letters (in any order): "a", "f", "h", "l", "m", "p", and "u".

Option letters that may follow the letter "-n" have the following meaning:

| Option | Description |
|---|---|
| a (After parent) | Ensures that each tag name is unique. If a COBOL data-name within the specified group item is not unique in the COBOL program file, the tag name is formed by recursively adding the sequence ".of." and the parent name after the data-name. For example, if there were a variable named "B" in a COBOL data structure named "A", the tag name would be "B.of.A". This is done until the tag name becomes unique. |
| f (First) | Specifies that the first letter of the tag name be capitalized. |
| h (Hyphen out) | Removes hyphen characters in the COBOL data-name from the tag name. |
| l (Lowercase) | Unless overridden by the "f" or "m" options, specifies that all characters in the tag name be represented in lowercase. |
| m (Mixed case) | Specifies that the first letter after a hyphen character in the COBOL data-name be represented as uppercase in the generated tag name. |
| p (Prefix out) | Removes all characters in the COBOL data-name up to and including the first hyphen. This option is useful where all data items in a structure begin with the same sequence. However, use this option with care. If the item name contains no hyphen characters, then the generated tag name will be empty. |
| u (Uppercase) | Specifies that all characters in the COBOL data-name be represented as uppercase in the generated tag name. |

## *Schema Options*

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

By default, a schema file is generated that will be used to validate an XML document. The schema file has the same base name as the other XML model files and has an extension of **.xsd**. Four formats of schema files are defined: XDR (BizTalk), XDR, Schema, and None.

A schema option is generated by entering a hyphen character (-) followed by the letter "s" and then by one of the following letters: "b", "d", "s", or "n".

Supported schema options include the following:

| Option | Description |
|---|---|
| -sb | The generated schema file complies with the older XDR (XML Data Reduced) schema format, with additions that make it compatible with BizTalk Mapper. |
| -sd | The generated schema file complies with the older XDR (XML Data Reduced) schema format. |
| -ss | The generated schema file complies with the current schema definition. (This is the default.) |
| -sn | No schema file is generated. |

# Referencing XML Model Files

XML model files (on page 28) may be referenced by the COBOL application by means of a traditional path name or by an Internet address. Examples of references to XML model files are shown in the following table:

| Filename | Type of Referencing |
|---|---|
| c:\myfiles\myapp.xml | Simple path name |
| \\mysystem\myfiles\myapp.xml | UNC (Universal Naming Convention) |
| http://myserver/myfiles/myapp.xml | URL (Universal Resource Locator) |

The **cobtoxml** utility generates up to four XML documents (called model files) for each data structure that is specified. Three of these files are used internally by the COBOL application while one is provided for information purposes only. These XML documents include the following:

- **Example file.** The example file (a file having the **.xml** extension) is an XML document that does not contain any text values. It is identical to the template file, except that the COBOL attributes have been removed. The **xmlif** library does not use the example file. The example file is provided as a reference to assist the developer in designing any external XSLT stylesheets that may be needed.
- **Template file.** The template file (a file having the **.xtl** extension) is an XML document that does not contain any text values. Each element contains several COBOL-like attributes that describe the data. The **xmlif** library uses the template file to generate an XML document.
- **Internal XSLT stylesheet file.** The internal stylesheet (a file having the **.xsl** extension) is an XSLT stylesheet. It adds COBOL-like attributes to an existing XML document. The **xmlif** library automatically applies the internal XSLT stylesheet when importing an XML document into COBOL. The internal XSLT stylesheet is applied after the external XSLT stylesheet where the XML IMPORT FILE/TEXT statement specifies the optional external XSLT stylesheet parameter.
- **Schema file.** The **xmlif** library uses the schema file (a file having the **.xsd** extension), if present, to validate the content of an imported XML data document. If the schema file is absent, no validation is performed.

  **Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

# Chapter 6: xmlif Library Reference

This chapter describes the **xmlif** library, which is used by XML Extensions for RM/COBOL at runtime.

## What is the xmlif Library?

The **xmlif** library is a 32-bit dynamic link library on Windows (**xmlif.dll**) or a shared object on UNIX (**xmlif.so**) that is callable from RM/COBOL object programs. It provides facilities to process, manipulate, and validate XML documents.

On Windows, the **xmlif** library uses the Microsoft MSXML 4.0 parser; on UNIX, the **xmlif** library uses the XML parser (libxml) and the XSLT transformation processor (libxslt) from the C libraries for the Gnome project. For additional information, see Installing XML Extensions (on page 11) and the "Deployment" section in XML Extensions Package (on page 10).

The statements used by the **xmlif** library are grouped into the following categories:

- Document Processing Statements (on page 64). These statements are used to process, manipulate, or validate XML documents.
- Document Management Statements (on page 72). These statements are used to copy an XML document from an external file to an internal text string and vice versa.
- Directory Management Statements (on page 75). These statements are useful when implementing directory-polling schemes.
- State Management Statements (on page 78). These statements are used to control the state or condition of the **xmlif** library.

**Note** Each statement contains zero or more positional parameters. These parameters are used to specify such items as the source or destination data item, source or destination XML document, model files produced by the **cobtoxml** utility, and flags. In some statements, trailing positional parameters are optional and may be omitted, as specified in the statement descriptions in this chapter.

# Document Processing Statements

Document processing statements are used to process, manipulate, or validate XML documents. They are grouped by function as follows:

- **Export statements.** These statements are available to convert the content of a COBOL data item to an XML document that may be represented as an external file or an internal text string. See XML EXPORT FILE (on page 64) and XML EXPORT TEXT (on page 66).
- **Import statements.** These statements are available to convert the content of an XML document—either an external file or an internal text string—to a COBOL data item. See XML IMPORT FILE (on page 67) and XML IMPORT TEXT (on page 68).
- **Test and validation statements.** These statements are available to verify that an XML document—either an external file or an internal text string—is well-formed or valid. They include:
  - XML TEST WELLFORMED-FILE (on page 69)
  - XML TEST WELLFORMED-TEXT (on page 70)
  - XML VALIDATE FILE (on page 71)
  - XML VALIDATE TEXT (on page 72)
- **Transformation statement.** Lastly, XML TRANSFORM FILE (on page 70) transforms an XML document in an external file into a new external file by applying an XSLT stylesheet. The resulting file may have almost any form, including XML, HTML, PDF, RTF, and so forth.

## XML EXPORT FILE

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DataItem* | The name of the COBOL data item that contains data to be exported. |
| *DocumentName* | The name of a file that will receive the exported XML document. |
| *ModelFileName* | The name of the set of XML files produced by the cobtoxml utility that describe the COBOL data item. For more information, see Model Files (on page 28). |
| [*StyleSheetName*] | Optional. The name of an external XSLT stylesheet that will be used to transform the generated XML document before it is stored. |
| [*DocumentPrefix*] | Optional. A literal or the name of a COBOL data item that contains a document prefix; for example, a document type definition (DTD), which is to be output between the XML header and the first element of the exported XML document. For more information, see Document Type Definition Support (on page 53). |

### *Description*

The XML EXPORT FILE statement exports the content of the COBOL data item indicated by the *DataItem* parameter. The content of the data item is converted to an XML document using one or more files indicated by the *ModelFileName* parameter. The output of this conversion is to the file specified by the *DocumentName* parameter. If the optional *StyleSheetName* parameter is present, the external XSLT stylesheet is used to transform the document after it has been generated but before it is stored in the data file.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

### *Examples*

#### Without an External XSLT Stylesheet:

```
XML EXPORT FILE
    MY-DATA-ITEM
    "MY-DOCUMENT"
    "MY-MODEL-FILE".
IF NOT XML-OK GO TO Z.
```

#### With an External XSLT Stylesheet:

```
XML EXPORT FILE
    MY-DATA-ITEM
    "MY-DOCUMENT.XML"
    "MY-MODEL-FILE"
    "MY-STYLE-SHEET"
IF NOT XML-OK GO TO Z.
```

#### Without an External XSLT Stylesheet and With a Document Prefix:

```
XML EXPORT FILE
    MY-DATA-ITEM
    "MY-DOCUMENT"
    "MY-MODEL-FILE"
    OMITTED          *> no stylesheet
    "<!DOCTYPE root [" &
        "<!ENTITY CURRENCY ""&#036;"">" &
        "]>".
IF NOT XML-OK GO TO Z.
```

# XML EXPORT TEXT

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DataItem* | The name of the COBOL data item that contains data to be exported. |
| *DocumentPointer* | The name of a COBOL pointer data item that will point to the generated XML document as a text string after successful completion of the statement. |
| *ModelFileName* | The name of the set of XML files produced by the cobtoxml utility that describe the COBOL data item. For more information, see Model Files (on page 28). |
| [*StyleSheetName*] | Optional. The name of an external XSLT stylesheet that will be used to transform the generated XML document before it is stored. |
| [*DocumentPrefix*] | Optional. A literal or the name of a COBOL data item that contains a document prefix, for example, a document type definition (DTD), which is to be output between the XML header and the first element of the exported XML document. For more information, see Document Type Definition Support (on page 53). |

## *Description*

The XML EXPORT TEXT statement exports the content of the COBOL data item indicated by the *DataItem* parameter. The content of the data item is converted to an XML document using one or more files indicated by the *ModelFileName* parameter, and then it is output as a text string. The address of the text string is placed in the COBOL pointer data item specified by the *DocumentPointer* parameter. If the optional *StyleSheetName* parameter is present, the external XSLT stylesheet is used to transform the document after it has been generated but before it is stored as a text string.

A block of memory is allocated to hold the generated XML document. The descriptor of this memory block overrides any existing address descriptor in the COBOL pointer data item. The COBOL application is responsible for releasing this memory when it is no longer needed by using XML FREE TEXT (on page 73).

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

## *Examples*

### Without an External XSLT Stylesheet:

```
XML EXPORT TEXT
    MY-DATA-ITEM
    MY-DOCUMENT-POINTER
    "MY-MODEL-FILE".
IF NOT XML-OK GO TO Z.
```

**With an External XSLT Stylesheet:**

```
XML EXPORT TEXT
    MY-DATA-ITEM
    MY-DOCUMENT-POINTER
    "MY-MODEL-FILE"
    "MY-STYLE-SHEET"
IF NOT XML-OK GO TO Z.
```

**Without an External XSLT Stylesheet and With a Document Prefix:**

```
XML EXPORT TEXT
    MY-DATA-ITEM
    "MY-DOCUMENT"
    "MY-MODEL-FILE"
    OMITTED          *> no stylesheet
    "<!DOCTYPE root [" &
        "<!ENTITY CURRENCY ""&#036;"">" &
        "]>".
IF NOT XML-OK GO TO Z.
```

# XML IMPORT FILE

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DataItem* | The name of the COBOL data item that is to receive the imported data. |
| *DocumentName* | The name of the file that contains the XML document to be imported. |
| *ModelFileName* | The name of the set of XML files produced by the cobtoxml utility that describe the COBOL data item. For more information, see Model Files (on page 28). |
| [*StyleSheetName*] | Optional. The name of an external XSLT stylesheet that will be used to transform the imported XML document before it is stored in the data item. |

## *Description*

The XML IMPORT FILE statement imports the content of the file indicated by the *DocumentName* parameter. If the optional *StyleSheetName* parameter is present, the external XSLT stylesheet is first used to transform the document. The content of the XML document is converted to COBOL format using one or more files specified by the *ModelFileName* parameter, including the internal XSLT stylesheet, and then is stored in the data item specified by the *DataItem* parameter.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

### *Examples*

#### Without an External XSLT Stylesheet:

```
XML IMPORT FILE
    MY-DATA-ITEM
    "MY-DOCUMENT"
    "MY-MODEL-FILE".
IF NOT XML-OK GO TO Z.
```

#### With an External XSLT Stylesheet:

```
XML IMPORT FILE
    MY-DATA-ITEM
    "MY-DOCUMENT.XML"
    "MY-MODEL-FILE"
    "MY-STYLE-SHEET"
IF NOT XML-OK GO TO Z.
```

## XML IMPORT TEXT

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DataItem* | The name of the COBOL data item that is to receive the imported data. |
| *DocumentPointer* | The name of a COBOL pointer data item that points to an XML document that is stored in memory as a text string. |
| *ModelFileName* | The name of the set of XML files produced by the cobtoxml utility that describe the COBOL data item. For more information, see Model Files (on page 28). |
| [*StyleSheetName*] | Optional. The name of an external XSLT stylesheet that will be used to transform the imported XML document before it is stored in the data item. |

### *Description*

The XML IMPORT TEXT statement imports the content of the text string indicated by the *DocumentPointer* parameter. If the optional *StyleSheetName* parameter is present, the external XSLT stylesheet is used to transform the document before being converted to COBOL data format. The content of the XML document is converted to COBOL format using one or more files specified by the *ModelFileName* parameter, including the internal XSLT stylesheet, and then is stored in the data item specified by the *DataItem* parameter.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

### Examples

#### Without an External XSLT Stylesheet:

```
XML IMPORT TEXT
    MY-DATA-ITEM
    MY-DOCUMENT-POINTER
    "MY-MODEL-FILE".
IF NOT XML-OK GO TO Z.
```

#### With an External XSLT Stylesheet:

```
XML IMPORT TEXT
    MY-DATA-ITEM
    MY-DOCUMENT-POINTER
    "MY-MODEL-FILE"
    "MY-STYLE-SHEET"
IF NOT XML-OK GO TO Z.
```

## XML TEST WELLFORMED-FILE

This statement has the following parameter:

| Parameter | Description |
|---|---|
| *DocumentName* | The name of the file that contains the XML document to be tested. |

### Description

The XML TEST WELLFORMED-FILE statement tests the XML document specified by the *DocumentName* parameter to see if it is well-formed. However, the content of the document may or may not be valid.

A well-formed XML document is one that conforms to XML syntax rules. A valid XML document has content that conforms to rules specified by an XML schema file.

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

### Example

```
XML TEST WELLFORMED-FILE
    "MY-DOCUMENT".
IF NOT XML-OK GO TO Z.
```

# XML TEST WELLFORMED-TEXT

This statement has the following parameter:

| Parameter | Description |
|---|---|
| *DocumentPointer* | The name of a COBOL pointer data item that points to an XML document that is stored in memory as a text string. |

## Description

The XML TEST WELLFORMED-TEXT statement tests the XML document specified by the *DocumentPointer* parameter to see if it is well-formed. However, the content of the document may or may not be valid.

A well-formed XML document is one that conforms to XML syntax rules. A valid XML document has content that conforms to rules specified by an XML schema file.

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

## Example

```
XML TEST WELLFORMED-TEXT
    "MY-DOCUMENT".
IF NOT XML-OK GO TO Z.
```

# XML TRANSFORM FILE

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *InputDocumentName* | The filename of the document to transform (the input document). |
| *StyleSheetName* | The filename of the XSLT stylesheet used for the transformation. |
| *OutputDocumentName* | The filename of the transformed document (the output document). |

## Description

The XML TRANSFORM FILE statement transforms the XML document specified by the *InputDocumentName* parameter using the XSLT stylesheet specified by the *StyleSheetName* parameter into a new document specified by the *OutputDocumentName* parameter. The new document may or may not be an XML document depending on the XSLT stylesheet.

**Note** Specifying the internal XSLT stylesheet, one of the model files (see page 28), for the *StyleSheetName* parameter can be used to test the internal XSLT stylesheet

transform, which is occasionally helpful in debugging problems with importing documents into COBOL.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

### *Example*

```
XML TRANSFORM FILE
    "MY-IN-DOCUMENT"
    "MY-STYLESHEET"
    "MY-OUT-DOCUMENT.
IF NOT XML-OK GO TO Z.
```

## XML VALIDATE FILE

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DocumentName* | The name of the file that contains the XML document to be tested. |
| *SchemaName* | The name of the schema file that will be used to validate the XML document specified in *DocumentName*. |

### *Description*

The XML VALIDATE FILE statement tests the XML document specified by the *DocumentName* parameter to see if it is well-formed and valid.

A well-formed XML document is one that conforms to XML syntax rules. A valid XML document has content that conforms to rules specified by an XML schema file.

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

**Note** In the Windows implementation of XML Extensions, the Microsoft XML parser 4.0 ignores the document type definition (DTD) when validating an XML document against a schema file. Thus, any entities declared in the DTD will not be defined and cannot be referenced. Any XML document that contains entity references, other than the predefined XML entities, must be transformed with an XSLT stylesheet prior to validation against a schema file when using the Microsoft XML parser 4.0 so that any non-predefined entity references are removed. Otherwise, the document will fail validation.

### *Example*

```
XML VALIDATE FILE
    "MY-DOCUMENT"
    "MY-SCHEMA".
IF NOT XML-OK GO TO Z.
```

## XML VALIDATE TEXT

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DocumentPointer* | The name of a COBOL pointer data item that points to an XML document that is stored in memory as a text string. |
| *SchemaName* | The name of the schema file that will be used to validate the XML document specified in *DocumentPointer*. |

### *Description*

The XML VALIDATE TEXT statement tests the XML document specified by the *DocumentPointer* parameter to see if it is well-formed and valid.

A well-formed XML document is one that conforms to XML syntax rules. A valid XML document has content that conforms to rules specified by an XML schema file.

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

**Note** In the Windows implementation of XML Extensions, the Microsoft XML parser 4.0 ignores the document type definition (DTD) when validating an XML document against a schema file. Thus, any entities declared in the DTD will not be defined and cannot be referenced. Any XML document that contains entity references, other than the predefined XML entities, must be transformed with an XSLT stylesheet prior to validation against a schema file when using the Microsoft XML parser 4.0 so that any non-predefined entity references are removed. Otherwise, the document will fail validation.

### *Example*

```
XML VALIDATE TEXT
    "MY-DOCUMENT"
    "MY-SCHEMA".
IF NOT XML-OK GO TO Z.
```

# Document Management Statements

A number of statements are available to copy an XML document from an external file to an internal text string and vice versa. These document management statements include the following:

# XML FREE TEXT

This statement has the following parameter:

| Parameter | Description |
|---|---|
| *DocumentPointer* | The name of a COBOL pointer data item that points to an XML document. |

### *Description*

The XML FREE TEXT statement releases the COBOL memory referred to by the COBOL pointer data item specified by the *DocumentPointer* parameter.

### *Example*

```
XML FREE TEXT
    MY-POINTER
IF NOT XML-OK GO TO Z.
```

# XML GET TEXT

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DocumentPointer* | The COBOL pointer data item that will point to the in-memory text after successful completion of the statement. |
| *DocumentName* | The filename of XML document containing the text to load into memory. |

### *Description*

The XML GET TEXT statement copies the content of an XML document from the file specified by the *DocumentName* parameter to COBOL memory. A block of memory is allocated to contain the document. The address and size of the memory block are returned in the *DocumentPointer* parameter.

When the program has finished using the in-memory document, a call to XML FREE TEXT (on page 73) should be made to release the allocated memory.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

### *Example*

```
XML GET TEXT
    MY-POINTER
    "MY-DOCUMENT".
IF NOT XML-OK GO TO Z.
```

# XML PUT TEXT

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DocumentPointer* | The COBOL pointer data item that points to the in-memory text. |
| *DocumentName* | The filename that will contain the XML document upon successful completion of the statement. |

## Description

The XML PUT TEXT statement copies the content of the in-memory XML document specified by the *DocumentPointer* parameter to the external file specified by the *DocumentName* parameter.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

## Example

```
XML PUT TEXT
    MY-POINTER
    "MY-DOCUMENT".
IF NOT XML-OK GO TO Z.
```

# XML REMOVE FILE

This statement has the following parameter:

| Parameter | Description |
|---|---|
| *FileName* | The name of file to be removed. |

## Description

The XML REMOVE FILE statement deletes the file specified by the *FileName* parameter. If the specified filename does not contain an extension, then **.xml** is appended to the name. If the file does not exist, no error is returned.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

## Example

```
XML REMOVE FILE
    MY-FILE-NAME.
IF NOT XML-OK GO TO Z.
```

# Directory Management Statements

This section describes the statements that are useful when implementing directory-polling schemes:

- XML FIND FILE (on page 76)
- XML GET UNIQUEID (on page 77)

Directory polling, as related to XML documents, allows two or more independent processes to pass XML documents between the processes. For example, one or more writer processes may place XML documents in a well-known directory (a well-known directory is a directory name that is known to all of the interested processes). Each XML document must have been given a unique name. A reader process finds, processes, and removes XML documents from the same well-known directory.

Directory polling may be used to communicate with Microsoft's BizTalk server and other message-driven communications systems. It is a technique that may also be used between various RM/COBOL applications.

The RM/COBOL runtime is not scalable in the traditional sense; however, scalability can be achieved by using multiple RM/COBOL runtime systems (preferably running on separate hardware platforms) on the same local area network (LAN). Each of these separate runtime systems can use directory polling (to a directory that is available on the network) as a means of improving throughput.

It is not feasible to use multiple reader processes on the same directory because the XML FIND FILE statement, invoked from separate processes, could find the same file. For the Windows implementation, a sample C language program (**DirSplit**) is provided that will poll a single directory and distribute files to subdirectories as they arrive. This will allow separate COBOL programs each to process a separate subdirectory.

**Note** The following problems have been encountered on Windows systems running the older FAT32 file system:

- When a program is adding XML document files to a directory concurrently with another program that is moving XML document files to different directory using the C library function **rename** or the Windows API function **MoveFile**, it is possible for the wrong file to be moved or for the file to be moved to the wrong location. This failure can occur without the participation of XML Extensions.
- When a large number of XML document files are written to a directory by XML Extensions using XML EXPORT FILE (on page 64), it is possible that files will not be placed in the directory and no error will be returned by the operating system either to XML Extensions or to the program issuing the statement. It appears that the FAT32 file system may be limited to 65,535 files per directory (at least under certain conditions). Furthermore, if long filenames are used, multiple directory entries may be needed for each filename, further reducing the number of files per directory.

For these reasons, Liant recommends that directory polling be used only on Windows NT-based systems (that is, those running NTFS). These NT-based systems include Windows NT, Windows 2000, and Windows XP. However, these NT-based systems also could be configured to run the older FAT32 file system.

# XML FIND FILE

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DirectoryName* | The name of the directory to check for XML documents (files ending with the .xml extension). |
| *FileName* | The name of one XML document (file ending with the .xml extension) that was found in the specified directory. |

## Description

The XML FIND FILE statement looks in the directory specified by the *DirectoryName* parameter for an XML document (a file with the **.xml** extension). If there are one or more XML documents in the specified directory, the name of one of the files will be returned in the *FileName* parameter.

If the statement succeeds (the condition `XML-IsSuccess` is true), the XML document specified by the *FileName* parameter may be processed by using XML IMPORT FILE (on page 67).

Before calling XML FIND FILE again (to process the next file), you must call XML REMOVE FILE (on page 74) to delete the XML document that was just processed. Otherwise, the next call to the XML FIND FILE statement may return the same file.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**. The condition `XML-IsDirectoryEmpty` will be true if the directory is empty.

## Example

```
FIND-DOCUMENT.
    PERFORM WITH TEST AFTER UNTIL 0 > 1
        XML FIND FILE
            "MY-DIRECTORY"
            MY-FILE-NAME
        IF XML-IsSuccess
            EXIT PERFORM
        END-IF
        IF XML-IsDirectoryEmpty
            CALL "C$DELAY" USING 0.1
        END-IF
        IF NOT XML-OK GO TO Z.
    END-PERFORM
*> Process found document
```

# XML GET UNIQUEID

This statement has the following parameter:

| Parameter | Description |
|---|---|
| *UniqueID* | The unique value returned by this statement is a string representation having the same format as an UUID (Universal Unique Identifier). The string is a series of hexadecimal digits with embedded hyphen characters. The string is enclosed in brace characters ({ and }). The entire string is 38 characters in length.<br><br>On Windows systems, the value is an actual UUID. On UNIX systems, the value is a string having the same format as an UUID, but constructed by an internal algorithm. This algorithm uses various components, including the system id, the start time of the run unit, the current time, and an internal counter, to generate a unique value. |

## *Description*

The XML GET UNIQUEID statement generates a unique identifier that may be used to form a unique filename. Please note that the return value might not contain any alphabetic characters. Therefore, it would be a good programming practice to add an alphabetic character to the name for those systems where filenames require at least one alphabetic character (see the following example).

This statement may be used in conjunction with the COBOL STRING statement to generate a unique filename.

A status value is returned in the `XML-data-group` data item, which is defined in the copy file, **lixmldef.cpy**.

## *Example*

```
MOVE SPACES TO MY-FILE-NAME.
XML GET UNIQUEID
    MY-UNIQUEID.
IF NOT XML-OK GO TO Z.
STRING "mydir\a"   DELIMITED BY SIZE
       MY-UNIQUEID DELIMITED BY SPACE
       ".xml"      DELIMITED BY SIZE
    INTO MY-FILE-NAME.
```

# State Management Statements

Calls to the following XML statements control several states or conditions of the **xmlif** library, including:

- **Initialization and termination.** Before issuing a call to any other **xmlif** library statement, XML INITIALIZE (on page 79) must be called. (If XML INITIALIZE has not been called, any subsequent calls, for example, XML EXPORT FILE, will fail.) Similarly, XML TERMINATE (on page 80) should be called when the COBOL application is finished using the **xmlif** library. (If XML TERMINATE has not been called prior to program termination, there are no consequences.)

- **Empty array occurrences.** As an optimization, trailing "empty" occurrences of arrays are normally not generated by the statements, XML EXPORT FILE (on page 64) or XML EXPORT TEXT (on page 66).

  An empty occurrence of an array is defined to be one where the numeric items have a zero value and the nonnumeric items have a value equivalent to all spaces. This is the default state and is equivalent to calling XML DISABLE ALL-OCCURRENCES (on page 80). It is possible to force all occurrences to be output by calling XML ENABLE ALL-OCCURRENCES (on page 81).

- **COBOL attributes.** For each element generated by the statements, XML EXPORT FILE (on page 64) or XML EXPORT TEXT (on page 66), there is a series of COBOL attributes that describe that element.

  The default state is not to output these attributes. However, it is sometimes necessary for a following activity (such as an XSLT stylesheet transformation) to have access to these attributes (specifically, length and subscript are often interesting to a follow-on activity). Using XML DISABLE ATTRIBUTES (on page 81) does not allow attributes to be written (this is the default). Using XML ENABLE ATTRIBUTES (on page 82) forces these attributes to be written.

- **Document caching.** XML documents, such as XSLT stylesheets, templates, and schemas, are normally considered to be static during the use of a production version of the application. That is, they are generated when the application is developed and are not modified until the application is modified.

  To optimize performance, when the **xmlif** library loads an XSLT stylesheet, a template, or a schema, the document is cached (that is, retained in memory) for an indefinite period of time. This is the default behavior. However, even in with the default behavior, a document in the cache may be flushed from memory if the cache is full and an XSLT stylesheet, template, or schema document not already in the cache is required for the current operation.

  If XSLT stylesheets, templates, or schemas are being generated dynamically, the user may selectively enable or disable caching. Executing XML ENABLE CACHE (on page 83), which sets the default behavior, enables caching of documents. Executing XML DISABLE CACHE (on page 82) disables caching, thus forcing all documents to be loaded each time they are referenced. Executing XML FLUSH CACHE (on page 83) flushes all documents from the cache without changing the state of caching (that is, if caching was enabled it remains enabled). Executing any of the following statements causes the contents of the cache to be flushed: XML INITIALIZE, XML ENABLE CACHE, XML DISABLE CACHE, XML FLUSH CACHE, and XML TERMINATE.

- **CodeBridge flags.** The data conversions performed by the statements, XML EXPORT FILE (on page 64), XML EXPORT TEXT (on page 66), XML IMPORT FILE (on page 67), and XML IMPORT TEXT (on page 68), use the CodeBridge library (which is built into the RM/COBOL runtime) to perform these conversions. By default, the following CodeBridge flags are set: PF_TRAILING_SPACES, PF_LEADING_SPACES, PF_LEADING_MINUS, and PF_ROUNDED.

  **Note** The CodeBridge flags are C macros. They are case sensitive and require the use of the underscore character.

  XML SET FLAGS (on page 86) is available to alter these defaults. Refer to the CodeBridge manual for a more complete presentation of the CodeBridge conversion library.

- **Internal character encoding.** Characters within alphanumeric data elements in a COBOL program are normally encoded using the conventions of underlying operating systems. Under some conditions, it may be desirable to encode these same data items using UTF-8 encoding. (UTF-8 is a format for representing Unicode.) XML SET ENCODING (on page 85) is provided to switch between the local encoding format and UTF-8.

  **Note** Both the UNIX and Windows implementations of XML Extensions allow the in-memory representation of element content to use UTF-8 encoding. This may be useful for COBOL applications that wish to pass UTF-8-encoded data to other processes. XML documents are normally encoded using Unicode. XML Extensions always generates UTF-8 data. For more information, see COBOL and Character Encoding (on page 40) and XML and Character Encoding (on page 53).

## XML INITIALIZE

This statement has no parameters.

### *Description*

The XML INITIALIZE statement opens a session with the **xmlif** library. It ensures that the RM/COBOL runtime system is the required version (8 or greater) and retrieves required information from the runtime system. RM/COBOL runtime version 8 or greater is required because information needed by the **xmlif** library is not available in prior runtime versions. The underlying XML parser is also initialized.

The execution of this statement cause the document cache to be flushed from memory.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**. Errors can occur if the library is already initialized, the RM/COBOL runtime version is not 8 or greater, or the underlying XML parser initialization fails.

### *Example*

```
XML INITIALIZE.
IF NOT XML-OK GO TO Z.
```

# XML TERMINATE

This statement has no parameters.

### Description

The XML TERMINATE statement flushes the document cache and closes a session with the **xmlif** library. The interface to the underlying XML parser is also closed. Any memory blocks that were allocated by the **xmlif** library are freed.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**. Errors can occur if the library is not currently initialized, the calls to free memory fail, or the underlying XML parser termination fails.

### Example

```
XML TERMINATE.
IF NOT XML-OK GO TO Z.
```

# XML DISABLE ALL-OCCURRENCES

This statement has no parameters.

### Description

The XML DISABLE ALL-OCCURRENCES statement causes unnecessary empty array occurrences not to be generated by the statements, XML EXPORT FILE (on page 64) and XML EXPORT TEXT (on page 66). An empty array is one in which all numeric elements have a zero value and all nonnumeric elements have a value of all spaces.

There is some interoperation with the statements, XML DISABLE ATTRIBUTES (on page 81) and XML ENABLE ATTRIBUTES (on page 82). If attributes are enabled (that is, XML ENABLE ATTRIBUTES has been called), then all empty occurrences are not generated. If attributes are disabled (the default state or if XML DISABLE ATTRIBUTES has been used), then all trailing empty occurrences are not generated. If attributes are enabled, then the subscript is present and so leading, or intermediate, empty occurrences are not needed as placeholders to ensure that the correct subscript is calculated.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

### Example

```
XML DISABLE ALL-OCCURRENCES.
IF NOT XML-OK GO TO Z.
```

# XML ENABLE ALL-OCCURRENCES

This statement has no parameters.

### Description

The XML ENABLE ALL-OCCURRENCES statement causes all occurrence of an array to be generated by the statements, XML EXPORT FILE (on page 64) and XML EXPORT TEXT (on page 66), regardless of the content of the array.

All occurrences of an array are generated regardless of whether attributes are enabled or disabled.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

### Example

```
XML ENABLE ALL-OCCURRENCES.
IF NOT XML-OK GO TO Z.
```

# XML DISABLE ATTRIBUTES

This statement has no parameters.

### Description

The XML DISABLE ATTRIBUTES statement causes the COBOL attributes of an XML element to be omitted from an exported XML document. This is the default state.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

### Example

```
XML DISABLE ATTRIBUTES.
IF NOT XML-OK GO TO Z.
```

## XML ENABLE ATTRIBUTES

This statement has no parameters.

### *Description*

The XML ENABLE ATTRIBUTES statement causes the COBOL attributes of an XML element to be generated in an exported XML document

Some of the COBOL attributes (such as length and subscript) may be useful to an external XSLT stylesheet.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

### *Example*

```
XML ENABLE ATTRIBUTES.
IF NOT XML-OK GO TO Z.
```

## XML DISABLE CACHE

This statement has no parameters.

### *Description*

The XML DISABLE CACHE statement disables the caching of XSLT stylesheets, templates, and schemas. Besides disabling caching, executing this statement also flushes the document cache.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

### *Example*

```
XML DISABLE CACHE.
IF NOT XML-OK GO TO Z.
```

# XML ENABLE CACHE

This statement has no parameters.

### Description

The XML ENABLE CACHE statement enables the caching of XSLT stylesheets, templates, and schemas, and flushes the document cache immediately, even if document caching was already enabled.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

### Example

```
XML ENABLE CACHE.
IF NOT XML-OK GO TO Z.
```

# XML FLUSH CACHE

This statement has no parameters.

### Description

The XML FLUSH CACHE statement flushes the cache of XSLT stylesheet, templates, and schema documents, and flushes the document cache. The enabled or disabled state of caching is not changed by this statement.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

### Example

```
XML FLUSH CACHE.
IF NOT XML-OK GO TO Z.
```

# XML GET STATUS-TEXT

This statement has no named parameters.

## Description

A non-successful termination of an XML statement may cause one or more lines of descriptive text to be placed in a queue. The XML GET STATUS-TEXT statement fetches the next line of descriptive text.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**. The following condition names are also described in this copy file:

- `XML-IsSuccess`. A successful completion occurred (no informative, warning, or error messages).
- `XML-OK`. An OK (or satisfactory) completion occurred, including informative or warning messages.
- `XML-IsDirectoryEmpty`. An informative status indicating that XML FIND FILE (on page 76) found no XML documents in the indicated directory.

An example of processing the status information in this item is found below and in the copy file, **lixmldsp.cpy**.

## Example

```
Display-Status.
    If Not XML-IsSuccess
        Perform With Test After Until XML-NoMore
            XML GET STATUS-TEXT
            Display XML-StatusText
        End-Perform
    End-If.
```

**Note** In the **lixmldef.cpy** copy file, the definition of the `XML-StatusText` field may be edited from the default of 80 to change the size of the buffer used to contain XML status information. See Displaying Status Information (on page 46).

# XML SET ENCODING

This statement has the following parameter:

| Parameter | Description |
|---|---|
| *Encoding* | The value of this parameter must be either “local” or “utf-8”. If the value is “local”, then the character encoding used by the operating system is used. If the value is “utf-8”, then the data is treated as UTF-8 encoded. The parameter value is case insensitive. Any hyphen and underscore characters are optional. For example, “LOCAL”, “Local”, and “local” are equivalent. “UTF-8”, “Utf_8”, and “utf8” are also equivalent. |

## *Description*

The XML SET ENCODING statement allows the developer to specify the character encoding of data within a COBOL data structure. The developer may use this statement to switch between the local character encoding and UTF-8.

**Note** If the value of the *Encoding* parameter specifies “utf-8”, the RM_ENCODING environment variable (on page 40) is ignored. For more information on this environment variable, see COBOL and Character Encoding (on page 40).

Although, the XML SET ENCODING statement does not affect the character encoding of the XML document, it does affect the character encoding of the data in the COBOL program. For more information, see Data Representation (on page 40).

The XML SET ENCODING statement returns an error status value if the value of the *Encoding* parameter is not recognized.

## *Example*

```
XML SET ENCODING "local".
IF NOT XML-OK GO TO EXIT-1.
```

The default value is “local”. If XML SET ENCODING is never called, the default is used.

# XML SET FLAGS

The statement has the following parameter:

| Parameter | Description |
|---|---|
| *Flags* | A numeric value that represents one or more flags. These flags are a subset of the flags defined for CodeBridge. |

## Description

The XML SET FLAGS statement sets some flag values that are used for internal data conversion. Valid flag values are specified in the copy file, **lixmldef.cpy**. The default flag setting is the OR of the following values: PF-Leading-Spaces, PF-Trailing-Spaces, PF-Leading-Minus and PF-Rounded.

**Note** These flag values are 78-level constants. They are case insensitive and require the use of the hyphen character.

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**.

## Example

```
XML SET FLAGS
   MY-FLAGS.
IF NOT XML-OK GO TO Z.
```

# Appendix A: XML Extensions Examples

This appendix contains a collection of programs or program fragments that illustrate how the **xmlif** library statements are used. These examples are tutorial in nature and offer useful techniques to help you become familiar with the basics of using XML Extensions for RM/COBOL. More examples can be found in the XML Extensions examples directory, **Examples**.

**Note** You will find it instructive to examine these examples first before referring to Appendix B: *XML Extensions Sample Application Programs* (on page 175), which describes how to use and access the more complete application programs that are included with the XML Extensions development system.

The following example programs are provided in this appendix:



Additionally, three batch files are provided to facilitate use of the example programs. See Example Batch Files (on page 173).

# Example 1: Export File and Import File

This program first writes (or exports) an XML document file from the content of a COBOL data item. Then the program reads (or imports) the same XML document and places the content in the same COBOL data item.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 67), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements. For more information, see Model Files (on page 28) and Chapter 5: *cobtoxml Utility Reference* (on page 57).

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example1.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example1 y` |
| 2 | `cobtoxml example1 Liant-Address` |
| 3 | `move /y example1.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example1.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example1 k` |

Line 1 compiles the **example1.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 1 object filename is **example1.cob**, and the model filenames are **example1.xml**, **example1.xtl**, **example1.xsl**, and **example1.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 1 object file, **example1.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example1.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application that opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which could cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

This COBOL program illustrates how an XML document is generated from a COBOL data item, and then how the content of an XML document may be converted into COBOL data format and stored in a COBOL data item.

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **liant1.xml** using the XML EXPORT FILE statement.

Next, the content of the XML document is imported from the file, **liant1.xml**, and placed in the same data item using the XML IMPORT FILE statement.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "8911 Capital of Texas Highway North".
    02  Address-2  Pic X(64) Value "Suite 4300".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78759.
    02  Time-Stamp Pic 9(8).
```

This data item stores company address information (in this case, Liant's). The last field of the item is a time stamp containing the time that the program was executed. This item is included to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

# Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example1.cbl**.

## Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT FILE`<br>`  Liant-Address`<br>`  "Liant1"`<br>`  "Example1".` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Importing an XML Document

| COBOL Statement | Description |
|---|---|
| `Move Spaces to Liant-Address.` | Ensure that the Liant-Address item contains no data. |
| `XML IMPORT FILE`<br>`  Liant-Address`<br>`  "Liant1"`<br>`  "Example1".` | Execute the XML IMPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `  Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `  Stop Run.` | Terminate the COBOL program. |
| `  Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named Z, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition XML-IsSuccess is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

# Execution Results for Example 1

The following sections display the output of the COBOL program that is run and the XML document that is generated.

## COBOL Display

**Note** Pressing a key will terminate the program.

Running the program (**runcobol example1**) produces the following display:

```
Example-1 - Illustrate EXPORT FILE and IMPORT FILE
Liant1.xml exported by XML EXPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
16273191
Liant1.xml imported by XML IMPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
16273191

You may inspect 'Liant1.xml'

Status: 0000
Press a key to terminate:
```

## XML Document

Microsoft Internet Explorer may be used to view the generated XML document, **liant1.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
    <name>Liant Software Corporation</name>
    <address-1>8911 Capital of Texas Highway North</address-1>
    <address-2>Suite 4300</address-2>
    <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78759</zip>
    </address-3>
    <time-stamp>16273191</time-stamp>
  </liant-address>
</root>
```

# Example 2: Export File and Import File with XSLT Stylesheets

This program first writes (or exports) an XML document file from the content of a COBOL data item. Then the program reads (or imports) the same XML document and places the content in the same COBOL data item.

This example is almost identical to Example 1: Export File and Import File (on page 88). However, an external XSLT stylesheet is used to transform the exported document into a different format. Similarly, when the document is imported, an external XSLT stylesheet is used to reformat the document into the form that is expected by COBOL. For more information on stylesheets, see XSLT Stylesheet Files (on page 55).

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 67), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

**Note** In this example, the XML EXPORT FILE and XML IMPORT FILE statements each contain an additional parameter: the name of the external XSLT stylesheet being used for the transformation.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements. For more information, see Model Files (on page 28) and Chapter 5: *cobtoxml Utility Reference* (on page 57).

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example2.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example2 y` |
| 2 | `cobtoxml example2 Liant-Address` |
| 3 | `move /y example2.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example2.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example2 k` |

Line 1 compiles the **example2.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 2 object filename is **example2.cob**, and the model filenames are **example2.xml**, **example2.xtl**, **example2.xsl** and **example2.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 2 object file, **example2.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example2.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application that opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

This COBOL program illustrates how an XML document is generated from a COBOL data item, and then how the content of an XML document may be converted into COBOL data format and stored in a COBOL data item.

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **liant2.xml** using the XML EXPORT FILE statement.

Next, the content of the XML document is imported from the file, **liant2.xml**, and placed in the same data item using the XML IMPORT FILE statement.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "8911 Capital of Texas Highway North".
    02  Address-2  Pic X(64) Value "Suite 4300".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78759.
    02  Time-Stamp Pic 9(8).
```

This data item stores company address information (in this case, Liant's). The last field of the structure is a time stamp containing the time that the program was executed. This item is included to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

# Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example2.cbl**.

## Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT FILE`<br>`  Liant-Address`<br>`  "Liant2"`<br>`  "Example2"`<br>`  toExt.` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>the model filename,<br>and the external XSLT stylesheet name. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Importing an XML Document

| COBOL Statement | Description |
|---|---|
| `Move Spaces to Liant-Address.` | Ensure that the Liant-Address structure contains no data. |
| `XML IMPORT FILE`<br>`  Liant-Address`<br>`  "Liant2"`<br>`  "Example2"`<br>`  toInt.` | Execute the XML IMPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>the model filename,<br>and the external XSLT stylesheet name. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `  Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `  Stop Run.` | Terminate the COBOL program. |
| `  Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named `Z`, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## XSLT Stylesheets for Example 2

The two external XSLT stylesheets used in this example are for reference only (a tutorial on XSLT stylesheet development is outside the scope of this manual). The first is contained in the file, **toExt.xsl**. It is used by the XML EXPORT FILE statement to transform the generated XML document to an external format. The second is contained in the file, **toInt.xsl**, and is used by the XML IMPORT FILE statement to transform the input XML document to match the COBOL format.

These external XSLT stylesheets are user-defined and manually generated using a text editor program. Other tools, such as Microsoft's BizTalk Mapper, may be used to generate external XSLT stylesheets.

### *toExt.xsl*

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="UTF-8" />
  <xsl:template match="/">
    <xsl:apply-templates select="root/liant-address" />
  </xsl:template>
  <xsl:template match="liant-address">
    <LiantAddress>
      <Information>
        <xsl:attribute name="Name">
          <xsl:value-of select="name/text()" />
        </xsl:attribute>
        <xsl:attribute name="Address1">
          <xsl:value-of select="address-1/text()" />
        </xsl:attribute>
        <xsl:attribute name="Address2">
          <xsl:value-of select="address-2/text()" />
        </xsl:attribute>
        <xsl:attribute name="City">
          <xsl:value-of select="address-3/city/text()" />
        </xsl:attribute>
        <xsl:attribute name="State">
          <xsl:value-of select="address-3/state/text()" />
        </xsl:attribute>
        <xsl:attribute name="Zip">
          <xsl:value-of select="address-3/zip/text()" />
        </xsl:attribute>
      </Information>
      <TimeStamp>
        <xsl:attribute name="Value">
          <xsl:value-of select="time-stamp/text()" />
        </xsl:attribute>
      </TimeStamp>
    </LiantAddress>
  </xsl:template>
</xsl:stylesheet>
```

### *toInt.xsl*

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="UTF-8" />
  <xsl:template match="/">
    <xsl:apply-templates select="LiantAddress" />
  </xsl:template>
  <xsl:template match="LiantAddress">
    <root>
      <liant-address>
        <name>
          <xsl:value-of select="Information/@Name" />
        </name>
        <address-1>
          <xsl:value-of select="Information/@Address1" />
        </address-1>
        <address-2>
          <xsl:value-of select="Information/@Address2" />
        </address-2>
        <address-3>
          <city>
            <xsl:value-of select="Information/@City" />
          </city>
          <state>
            <xsl:value-of select="Information/@State" />
          </state>
          <zip>
            <xsl:value-of select="Information/@Zip" />
          </zip>
        </address-3>
        <time-stamp>
          <xsl:value-of select="TimeStamp/@Value" />
        </time-stamp>
      </liant-address>
    </root>
  </xsl:template>
</xsl:stylesheet>
```

## Execution Results for Example 2

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

**Note** Pressing a key will terminate the program.

Running the program (**runcobol example2**) produces the following display:

```
Example-2 - Illustrate EXPORT FILE and IMPORT FILE with XSLT stylesheets
Liant2.xml exported by XML EXPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
10415057
Liant2.xml imported by XML IMPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
10415057

You may inspect 'Liant2.xml'

Status: 0000
Press a key to terminate:
```

### *XML Document*

Microsoft Internet Explorer may be used to view the generated XML document, **liant2.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
<LiantAddress>
  <Information Name="Liant Software Corporation" Address1="8911 Capital of
   Texas Highway North" Address2="Suite 4300" City="Austin" State="TX"
   Zip="78759" />
  <TimeStamp Value="10415057" />
</LiantAddress>
```

This XML document differs from the document generated in Example 1: Export File and Import File (on page 88). Items that were shown as individual data elements in Example 1 are now shown as attributes of higher-level elements. Notice that this document contains no text. All of the information is contained in the markup.

# Example 3: Export File and Import File with OCCURS DEPENDING

This program first writes (or exports) an XML document file from the content of a COBOL data item. Then the program reads (or imports) the same XML document and places the content in the same COBOL data item.

This program is very similar to Example 1: Export File and Import File (on page 88). However, the data item has been modified so that an OCCURS DEPENDING clause is present.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 67), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example3.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example3 y` |
| 2 | `cobtoxml example3 Liant-Address` |
| 3 | `move /y example3.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example3.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example3 k` |

Line 1 compiles the **example3.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 3 object filename is **example3.cob**, and the model filenames are **example3.xml**, **example3.xtl**, **example3.xsl**, and **example3.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 3 object file, **example3.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example3.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application that opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

This COBOL program illustrates how an XML document is generated from a COBOL data item, and then how the content of an XML document may be converted into COBOL data format and stored in a COBOL data item.

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **liant3.xml** using the XML EXPORT FILE statement.

Next, the content of the XML document is imported from the file, **liant3.xml**, and placed in the same data structure using the XML IMPORT FILE statement.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the copy file, **liant3.cpy**, is as follows:

```
01  Liant-Address.
    02  Time-Stamp    Pic 9(8).
    02  Name          Pic X(64)
            Value "Liant Software Corporation".
    02  City          Pic X(32) Value "Austin".
    02  State         Pic X(2) Value "TX".
    02  Zip           Pic 9(5) Value 78759.
    02  Address-Lines Pic 9.
    02  Address-Line  Pic X(64)
            Occurs 1 to 5 times
            Depending on Address-Lines.
```

This data item stores company address information (in this case, Liant's). This structure differs from Example 1: Export File and Import File in that an OCCURS DEPENDING phrase has been added to the structure. Instead of having separate data-names for `Address-1` and `Address-2`, a variable length array named `Address-Line` has been defined. Since Address-Line is variable length, it must be the last data item in the structure. A new data item named `Address-Lines` has been added just prior to the Address-Line array. Address-Lines is the depending variable for the array Address-Line.

The first field of the structure is a time stamp containing the time that the program was executed. This item is included to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the COBOL program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example3.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT FILE`<br>`  Liant-Address`<br>`  "Liant3"`<br>`  "Example3".` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Importing an XML Document

| COBOL Statement | Description |
|---|---|
| `Move Spaces to Liant-Address.` | Ensure that the Liant-Address structure contains no data. |
| `XML IMPORT FILE`<br>`  Liant-Address`<br>`  "Liant3"`<br>`  "Example3".` | Execute the XML IMPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the "Termination Test Logic" table). |
| `Stop Run.` | Terminate the COBOL program. |
| `Copy "lixmldsp.cpy".` | Copy in the status display logic (see the "Status Display Logic" table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named `Z`, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example 3

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

**Note** Pressing a key will terminate the program.

Running the program (**runcobol example3**) produces the following display:

```
Example-3 - Illustrate EXPORT FILE and IMPORT FILE with OCCURS DEPENDING
Liant3.xml exported by XML EXPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
13313414
Liant3.xml imported by XML IMPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
13313414

You may inspect 'Liant3.xml'

Status: 0000
Press a key to terminate:
```

### *XML Document*

Microsoft Internet Explorer may be used to view the generated XML document, **Liant3.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
    <time-stamp>13313414</time-stamp>
    <name>Liant Software Corporation</name>
    <city>Austin</city>
    <state>TX</state>
    <zip>78759</zip>
    <address-lines>2</address-lines>
    <address-line>8911 Capital of Texas Highway North</address-line>
    <address-line>Suite 4300</address-line>
  </liant-address>
</root>
```

# Example 4: Export File and Import File with Sparse Arrays

This example illustrates how the **xmlif** library may work with sparse arrays. The **xmlif** library distinguishes between an empty occurrence and a non-empty occurrence. An occurrence is an empty occurrence when all of its numeric elementary data items have a zero value and all of its nonnumeric elementary data items contain spaces; otherwise, the occurrence is a non-empty occurrence. A sparse array is an array that contains a combination of empty and non-empty occurrences. Empty occurrences need not be exported unless they are needed to locate (determine the subscript) of a subsequent non-empty occurrence. Normally, this means that trailing empty occurrences, that is, a contiguous series of empty occurrences at the end of the array, are not exported. Sparse arrays may also be imported.

This program first writes (or exports) several XML document files from the content of a COBOL data item (using various combinations of the XML ENABLE ATTRIBUTES, XML DISABLE ATTRIBUTES, XML ENABLE ALL-OCCURRENCES, and XML DISABLE ALL-OCCURRENCES statements). Then the program reads (or imports) the same XML documents (plus a couple of pre-existing documents) and places the content in the same COBOL data item.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 67), which reads an XML document (from a file) into a COBOL data item.
- XML ENABLE ATTRIBUTES (on page 82), which causes exported XML document to contain descriptive (COBOL-oriented) attributes.

  **Note** Although the default is not to add descriptive attributes to an XML document (see XML DISABLE ATTRIBUTES in the next item), among the attributes that may be added is the “subscript” attribute. This attribute contains the one-relative index of the occurrence within the array. When an XML document is imported, this subscript attribute is used (if present) to place the occurrence correctly within the array. If the subscript attribute is not present, then occurrences are assumed to occur sequentially.
- XML DISABLE ATTRIBUTES (on page 81), which causes exported XML documents not to contain descriptive attributes.

  **Note** The default is not to add descriptive attributes to an XML document.
- XML ENABLE ALL-OCCURRENCES (on page 81), which causes all occurrences of a data item to be exported to an XML document.
- XML DISABLE ALL-OCCURRENCES (on page 80), which causes only certain occurrences to be exported to the XML document.

  **Note** The default is to export only certain occurrences to the XML document.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example4.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example4 y` |
| 2 | `cobtoxml example4 Data-Table -sn` |
| 3 | `move /y example4.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example4.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example4 k` |

Line 1 compiles the **example4.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 4 object filename is **example4.cob**, and the model filenames are **example4.xml**, **example4.xtl**, and **example4.xsl**). The **-sn** (schema none) option on the **cobtoxml** utility disables the generation of a schema file, which is normally used to validate the content of an XML document. Note that under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 4 object file, **example4.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example4.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application and opens a separate window when executed from DOS. The `start /w` part of

the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

This COBOL program illustrates how several similar XML documents are generated from a single COBOL data item. It also illustrates how the content of several similar XML documents may be converted into COBOL data format and stored in a COBOL data item.

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Data-Table` (as defined in the copy file, **liant.cpy**) to several XML documents with the filenames of **table1.xml**, **table2.xml**, **table3.xml**, and **table4.xml** using the XML EXPORT FILE statement. Various combinations of the XML ENABLE ATTRIBUTES, XML DISABLE ATTRIBUTES, XML ENABLE ALL-OCCURRENCES, and XML DISABLE ALL-OCCURRENCES statements are used to alter the content of the generated XML documents.

Next, the content of these four XML documents (plus two additional "pre-created" XML documents, **table5.xml** and **table6.xml**) is imported and placed in the same data item using the XML IMPORT FILE statement. This example does not use a schema file to validate the input because the array is fixed size and not all of the XML documents that will be input contain all of the occurrences of the array. These XML documents and their content are described in Execution Results for Example 4 (on page 114).

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Data-Table.
    02              Value "[".
    02  Table-1     Occurs 6.
        03  X       Pic X.
        03  N       Pic 9.
    02              Value "]".
```

This data item contains an array with six occurrences. Each occurrence consists of a one-character, nonnumeric data item followed by a one-digit numeric data item. Note that the structure also contains two FILLER data items: the left brace ([) character at the beginning and the right brace(]) character at the end. The values of the FILLER data items are output as text in the XML document without associated tags.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example4.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `XML ENABLE ATTRIBUTES`<br>`If Not XML-OK Go To Z.`<br>`XML ENABLE All-OCCURRENCES`<br>`If Not XML-OK Go To Z.` | Selectively ENABLE or DISABLE ATTRIBUTES and ALL-OCCURRENCES. |
| `Initialize Data-Table.`<br>`Move "B" to X (2).`<br>`Move  2  to N (2).`<br>`Move "D" to X (4).`<br>`Move  4  to N (4).` | Initialize the Data-Table structure to the preferred values. |
| `XML EXPORT FILE`<br>`Data-Table`<br>`"Table1"`<br>`"Example4".` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename (Table1 – Table4),<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Importing an XML Document

| COBOL Statement | Description |
|---|---|
| `Initialize Data-Table.` | Ensure that the data item contains no data. |
| `XML IMPORT FILE`<br>`Data-Table`<br>`"Table1"`<br>`"Example4".` | Execute the XML IMPORT FILE statement specifying:<br>the data item address,<br>the XML document filename (Table1 – Table6),<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `Stop Run.` | Terminate the COBOL program. |
| `Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named Z, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition XML-IsSuccess is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example 4

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

**Note** Pressing a key will terminate the program.

Running the program (**runcobol example4**) produces the following display:

```
Example-4 - Illustrate EXPORT FILE and IMPORT FILE with sparse arrays
Table1.xml exported by XML EXPORT FILE: [ 0B2 0D4 0 0]
Table2.xml exported by XML EXPORT FILE: [ 0B2 0D4 0 0]
Table3.xml exported by XML EXPORT FILE: [ 0B2 0D4 0 0]
Table4.xml exported by XML EXPORT FILE: [ 0B2 0D4 0 0]
Table1.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
Table2.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
Table3.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
Table4.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
Table5.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
Table6.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]

You may inspect 'Table1.xml' - 'Table6.xml'

Status: 0000
Press a key to terminate:
```

### *XML Documents*

Microsoft Internet Explorer may be used to view the XML documents that are associated with this example. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

The files **table1.xml**, **table2.xml**, **table3.xml**, and **table4.xml** are generated with XML EXPORT FILE statements. All of these documents were generated from the same COBOL content. The files, **table5.xml** and **table6.xml**, which are supplied with the example, describe the same COBOL content.

The only non-empty occurrences are for the second and fourth elements of the array. The content of the six files should appear as follows.

#### Table1.xml

The XML DISABLE ATTRIBUTES and XML DISABLE ALL-OCCURRENCES statements are used to determine the content of this file. Trailing empty occurrences are deleted. However, some empty occurrences were generated so that the two non-empty occurrences are positioned correctly.

This example also uses FILLER data items. The left brace ([) and right brace(]) characters were defined within the data item as FILLER. The text associated with the FILLER is placed in the XML document without any tags.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <data-table>
    [
    <table-1>
      <x />
      <n>0</n>
    </table-1>
    <table-1>
      <x>B</x>
      <n>2</n>
    </table-1>
    <table-1>
      <x />
      <n>0</n>
    </table-1>
    <table-1>
      <x>D</x>
      <n>4</n>
    </table-1>
    ]
  </data-table>
</root>
```

### Table2.xml

The XML ENABLE ATTRIBUTES and XML DISABLE ALL-OCCURRENCES statements are used to determine the content of this file. Since each non-empty occurrence now contains a subscript attribute, none of the empty occurrences are generated.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root type="nonnumeric" kind="GRP">
  <data-table type="nonnumeric" kind="GRP" length="14" offset="4" id="1514">
    [
    <table-1 type="nonnumeric" kind="GRP" length="2" offset="5" minOccurs="6"
     maxOccurs="6" span="2" subscript="2" id="1558">
      <x type="nonnumeric" kind="ANS" length="1" offset="5" subscript="2"
       id="1580">B</x>
      <n type="numeric" kind="NSU" length="1" offset="6" scale="0"
       precision="1" subscript="2" id="1602">2</n>
    </table-1>
    <table-1 type="nonnumeric" kind="GRP" length="2" offset="5" minOccurs="6"
     maxOccurs="6" span="2" subscript="4" id="1558">
      <x type="nonnumeric" kind="ANS" length="1" offset="5" subscript="4"
       id="1580">D</x>
      <n type="numeric" kind="NSU" length="1" offset="6" scale="0"
       precision="1" subscript="4" id="1602">4</n>
    </table-1>
    ]
  </data-table>
</root>
```

### Table3.xml

The XML DISABLE ATTRIBUTES and XML ENABLE ALL-OCCURRENCES statements are used to determine the content of this file. These statements cause all occurrences, whether empty or non-empty, to be generated.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <data-table>
    [
    <table-1>
      <x />
      <n>0</n>
    </table-1>
    <table-1>
      <x>B</x>
      <n>2</n>
    </table-1>
    <table-1>
      <x />
      <n>0</n>
    </table-1>
    <table-1>
      <x>D</x>
      <n>4</n>
    </table-1>
    <table-1>
      <x />
      <n>0</n>
    </table-1>
    <table-1>
      <x />
      <n>0</n>
    </table-1>
    ]
  </data-table>
</root>
```

### Table4.xml

The XML ENABLE ATTRIBUTES and XML ENABLE ALL-OCCURRENCES statements are used to determine the content of this file. These statements produce the most verbose listing of occurrences possible. Every occurrence is listed with its attributes.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root type="nonnumeric" kind="GRP">
  <data-table type="nonnumeric" kind="GRP" length="14" offset="4" id="1514">
    [
    <table-1 type="nonnumeric" kind="GRP" length="2" offset="5" minOccurs="6"
     maxOccurs="6" span="2" subscript="1" id="1558">
      <x type="nonnumeric" kind="ANS" length="1" offset="5" subscript="1"
       id="1580" />
      <n type="numeric" kind="NSU" length="1" offset="6" scale="0"
       precision="1" subscript="1" id="1602">0</n>
    </table-1>
    <table-1 type="nonnumeric" kind="GRP" length="2" offset="5" minOccurs="6"
     maxOccurs="6" span="2" subscript="2" id="1558">
      <x type="nonnumeric" kind="ANS" length="1" offset="5" subscript="2"
       id="1580">B</x>
      <n type="numeric" kind="NSU" length="1" offset="6" scale="0"
       precision="1" subscript="2" id="1602">2</n>
    </table-1>
    <table-1 type="nonnumeric" kind="GRP" length="2" offset="5" minOccurs="6"
     maxOccurs="6" span="2" subscript="3" id="1558">
      <x type="nonnumeric" kind="ANS" length="1" offset="5" subscript="3"
       id="1580" />
      <n type="numeric" kind="NSU" length="1" offset="6" scale="0"
       precision="1" subscript="3" id="1602">0</n>
    </table-1>
    <table-1 type="nonnumeric" kind="GRP" length="2" offset="5" minOccurs="6"
     maxOccurs="6" span="2" subscript="4" id="1558">
      <x type="nonnumeric" kind="ANS" length="1" offset="5" subscript="4"
       id="1580">D</x>
      <n type="numeric" kind="NSU" length="1" offset="6" scale="0"
       precision="1" subscript="4" id="1602">4</n>
    </table-1>
    <table-1 type="nonnumeric" kind="GRP" length="2" offset="5" minOccurs="6"
     maxOccurs="6" span="2" subscript="5" id="1558">
      <x type="nonnumeric" kind="ANS" length="1" offset="5" subscript="5"
       id="1580" />
      <n type="numeric" kind="NSU" length="1" offset="6" scale="0"
       precision="1" subscript="5" id="1602">0</n>
    </table-1>
    <table-1 type="nonnumeric" kind="GRP" length="2" offset="5" minOccurs="6"
     maxOccurs="6" span="2" subscript="6" id="1558">
      <x type="nonnumeric" kind="ANS" length="1" offset="5" subscript="6"
       id="1580" />
      <n type="numeric" kind="NSU" length="1" offset="6" scale="0"
       precision="1" subscript="6" id="1602">0</n>
      </table-1>
    ]
  </data-table>
</root>
```

### Table5.xml

This file was manually generated using a text editor program in order to contain the minimum amount of information possible. Of all the attributes, only the subscript attribute is included. This allows all empty occurrences to be suppressed. In practice, an XSLT stylesheet or other software could generate this kind of document.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <data-table>
    [
    <table-1 subscript="2">
      <x>B</x>
      <n>2</n>
    </table-1>
    <table-1 subscript="4">
      <x>D</x>
      <n>4</n>
    </table-1>
    ]
  </data-table>
</root>
```

### Table6.xml

The only difference between this file and **table5.xml** is that the subscript reference has been moved from the occurrence level down to an element within the occurrence.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <data-table>
    [
    <table-1>
      <x subscript="2">B</x>
      <n>2</n>
    </table-1>
    <table-1>
      <x subscript="4">D</x>
      <n>4</n>
    </table-1>
    ]
  </data-table>
</root>
```

# Example 5: Export Text and Import Text

This program first writes (or exports) an XML document as a text string from the content of a COBOL data item. Then the program reads (or imports) the same XML document and places the content in the same COBOL data item. Finally, the text string representation of the XML document is copied to a disk file and the memory block that it occupied is released.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT TEXT (on page 66), which constructs an XML document (as a text string) from the content of a COBOL data item.
- XML IMPORT TEXT (on page 68), which reads an XML document (from a text string) into a COBOL data item.
- XML PUT TEXT (on page 74), which copies an XML document from a text string to a data file.
- XML FREE TEXT (on page 73), which releases the memory that was allocated by XML EXPORT TEXT to hold the XML document as a text string.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example5.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example5 y` |
| 2 | `cobtoxml example5 Liant-Address` |
| 3 | `move /y example5.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example5.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example5 k` |

Line 1 compiles the **example5.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 5 object filename is **example5.cob**, and the model filenames are **example5.xml**, **example5.xtl**, **example5.xsl**, and **example5.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 5 object file, **example5.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example5.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application and opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

This COBOL program illustrates how an XML document is generated from a COBOL data item, and then how the content of an XML document may be converted into COBOL data format and stored in a COBOL data item. This program is similar to Example 1: Export File and Import File (on page 88), except that the XML document is stored as a text string instead of a disk file.

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document as defined by the variable, *`Document-Pointer`*, using the XML EXPORT TEXT statement.

Next, the content of the XML document is imported from the file, **liant5.xml**, and placed in the same data item using the XML IMPORT TEXT statement.

Then, the contents of the text string are written to a disk file using the XML PUT TEXT statement. The memory block is deallocated using the XML FREE TEXT statement. The primary aim of using the XML PUT TEXT statement is to make the content of the XML document available as an external file for viewing.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "8911 Capital of Texas Highway North".
    02  Address-2  Pic X(64) Value "Suite 4300".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78759.
    02  Time-Stamp Pic 9(8).
```

This data item stores company address information (in this case, Liant's). The last field of the structure is a time stamp containing the time that the program was executed. The reason for this item is to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT TEXT statement returns a value in the `XML-Status`

field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example5.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT TEXT`<br>`  Liant-Address`<br>`  "Document-Pointer"`<br>`  "Example5".` | Execute the XML EXPORT TEXT statement specifying:<br>the data item address,<br>the XML document pointer name,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Importing an XML Document

| COBOL Statement | Description |
|---|---|
| `Move Spaces to Liant-Address.` | Ensure that the Liant-Address structure contains no data. |
| `XML IMPORT TEXT`<br>`  Liant-Address`<br>`  "Document-Pointer"`<br>`  "Example5".` | Execute the XML IMPORT TEXT statement specifying:<br>the data item address,<br>the XML document pointer name,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Copying an XML Document to a File

| COBOL Statement | Description |
|---|---|
| `XML PUT TEXT`<br>`  Document-Pointer`<br>`  "Liant5".` | Execute the XML PUT TEXT statement specifying:<br>the XML document pointer name<br>and the XML document filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Releasing the XML Document Memory

| COBOL Statement | Description |
|---|---|
| `XML FREE TEXT Document-Pointer.` | Execute the XML FREE TEXT statement specifying the XML document pointer name. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `Stop Run.` | Terminate the COBOL program. |
| `Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named `Z`, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution “falls through” to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example 5

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

**Note** Pressing a key will terminate the program.

Running the program (**runcobol example5**) produces the following display:

```
Example-5 - Illustrate EXPORT TEXT and IMPORT TEXT
Document exported by XML EXPORT TEXT
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
11232232
Document imported by XML IMPORT TEXT
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
11232232
Document memory written by XML PUT TEXT
Document memory released by XML FREE TEXT

You may inspect 'Liant5.xml'

Status: 0000
Press a key to terminate:
```

### XML Document

Microsoft Internet Explorer may be used to view the generated XML document, **liant5.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
    <name>Liant Software Corporation</name>
    <address-1>8911 Capital of Texas Highway North</address-1>
    <address-2>Suite 4300</address-2>
    <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78759</zip>
    </address-3>
    <time-stamp>11232232</time-stamp>
  </liant-address>
</root>
```

# Example 6: Export File and Import File with Directory Polling

This COBOL program illustrates how a series of XML documents may be placed in a specific directory and how directory polling may be used to process XML documents as they arrive in that specified directory. For more information on directory-polling schemes, see Directory Management Statements (on page 75).

The program first writes (or exports) five XML document files from the content of a COBOL data item. Each document has a unique name and is written to the same directory. Then the program polls the directory looking for an XML document. When one is found, the program reads (or imports) each XML document and places the content in the COBOL data item.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 67), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.
- XML GET UNIQUEID (on page 77), which is used to generate a unique identifier that can be used to form a filename.
- XML FIND FILE (on page 76), which finds a XML document file in the specified directory (if one is available).
- XML REMOVE FILE (on page 74), which deletes a file.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example6.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example6 y` |
| 2 | `cobtoxml example6 Time-Stamp` |
| 3 | `move /y example6.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example6.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example6 k` |

Line 1 compiles the **example6.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 6 object filename is **example6.cob**, and the model filenames are **example6.xml**, **example6.xtl**, **example6.xsl**, and **example6.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 6 object file, **example6.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example6.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application and opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted,

line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

The current time, which will become the content of an XML document, is recorded in a COBOL data item. Note that for this example, an elementary data item is used instead of a data item.

Because the name of each file within a directory must be unique, a unique filename is generated using the XML GET UNIQUEID statement. The returned value is combined with other text strings to form a path name using the STRING statement. The current time is placed in the `Time-Stamp` data item using the ACCEPT FROM TIME statement. The XML EXPORT FILE statement is used to output the data item as an XML document. This sequence is repeated until five XML documents have been placed in the specified directory.

Next, the program goes into a loop polling the specified directory. The XML FIND FILE statement is used. If the return status is `XML-IsSuccess`, then a file has been found and the program proceeds to process the file. If the return status is `XML-IsDirectoryEmpty`, then the directory is empty and the program issues a slight delay and then re-issues the XML FIND FILE statement. Any other status indicates an error.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the example, which in this case, is a single data item, is as follows:

```
01  Time-Stamp                  Pic 9(8).
```

This data item stores a time stamp acquired by using the ACCEPT FROM TIME statement.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
     03  XML-Status             PIC 9(4).
         88  XML-IsSuccess      VALUE XML-Success.
         88  XML-OK             VALUE XML-Success
                                THROUGH XML-StatusNonFatal.
         88  XML-IsDirectoryEmpty
                                VALUE XML-InformDirectoryEmpty.
     03  XML-StatusText         PIC X(80).
     03  XML-MoreFlag           PIC 9 BINARY(1).
         88 XML-NoMore          VALUE 0.
     03  XML-UniqueID           PIC X(40).
     03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example6.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting XML Documents with Unique Names

| COBOL Statement | Description |
|---|---|
| `XML GET UNIQUEID`<br>`  Unique-Name`<br>`If Not XML-OK Go To Z.` | Generate a unique identifier.<br>If the statement terminates unsuccessfully, go to the termination logic. |
| `Move Spaces to Unique-File-Name`<br>`String "Stamp\A" delimited by size`<br>`  Unique-Name delimited by SPACE`<br>`  ".xml" delimited by size`<br>`  into Unique-File-Name.` | Convert the unique identifier into a path name. |
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT FILE`<br>`  Liant-Address`<br>`  "Liant6"`<br>`  "Example6".` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Importing XML Documents by Directory Polling

| COBOL Statement | Description |
|---|---|
| `Perform Until 0 > 1` | Outer perform loop. Iterate until Exit Perform. |
| `  Perform Compute-Curr-Time` | The paragraph Compute-Curr-Time ACCEPTs the current time and converts it to an integer value. |
| `  Compute Stop-Time`<br>`      = Curr-Time + 100` | Compute Stop-Time to be 1 second after current time. |
| `  Perform Until 0 > 1` | Inner perform loop. Iterate until Exit Perform. |
| `    XML FIND FILE`<br>`      "Stamp"`<br>`       Unique-File-Name` | Execute XML FIND FILE parameters:<br>directory name<br>and filename. |
| `    If XML-IsSuccess`<br>`      Exit Perform`<br>`    End-If` | If the statement returned success,<br>exit the paragraph. |
| `    If XML-IsDirectoryEmpty`<br>`      Perform Compute-Curr-Time`<br>`      If Curr-Time > Stop-Time`<br>`        Exit Perform`<br>`      End-If`<br>`      Call "C$DELAY" Using 0.1`<br>`    End-If` | If the statement returns directory empty,<br>compute new current time, and<br>if the current time is greater than the stop time,<br>exit the perform.<br><br>Otherwise, do a short time delay. |
| `    If Not XML-OK`<br>`      Go To Z`<br>`    End-If` | If the statement terminates unsuccessfully,<br>go to the termination logic. |
| `  End-Perform` | The end of the inner perform loop. |
| `  If Curr-Time > Stop-Time`<br>`      Exit Perform`<br>`  End-If` | Check to see if the outer perform loop should terminate. |
| `  XML IMPORT FILE`<br>`     Time-Stamp`<br>`      Unique-File-Name`<br>`      "Example6"` | Import the file that was found using:<br>the data item,<br>the filename,<br>and the model filename. |
| `  If Not XML-OK Go To Z`<br>`      End-If` | If the statement terminates unsuccessfully, go to the termination logic. |
| `  XML REMOVE FILE`<br>`      Unique-File-Name` | Remove the file that has just been processed;<br>otherwise, find it again. |
| `  If Not XML-OK Go To Z`<br>`      End-If` | If the statement terminates unsuccessfully, go to the termination logic. |
| `End-Perform` | The end of the outer perform loop. |

## Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `    Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `    Stop Run.` | Terminate the COBOL program. |
| `    Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named Z, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition XML-IsSuccess is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example 6

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

Running the program (**runcobol example6**) produces two displays. The first display occurs after exporting five documents to the **Stamp** directory. The second display takes place after polling the **Stamp** directory and importing the five documents.

### First Display

**Note** Pressing a key will cause the program to continue.

```
Example-6 - Illustrate EXPORT FILE and IMPORT FILE with directory polling
Stamp\A{b8a405c0-d552-11d6-adbf-00a0cc274748}.xml exported by XMLExport
Contents: 15303258
Stamp\A{b8a405c2-d552-11d6-adbf-00a0cc274748}.xml exported by XMLExport
Contents: 15303264
Stamp\A{b8a405c4-d552-11d6-adbf-00a0cc274748}.xml exported by XMLExport
Contents: 15303264
Stamp\A{b8a405c6-d552-11d6-adbf-00a0cc274748}.xml exported by XMLExport
Contents: 15303264
Stamp\A{b8a405c8-d552-11d6-adbf-00a0cc274748}.xml exported by XMLExport
Contents: 15303264

You may display the 'Stamp' directory

Press a key to continue:
```

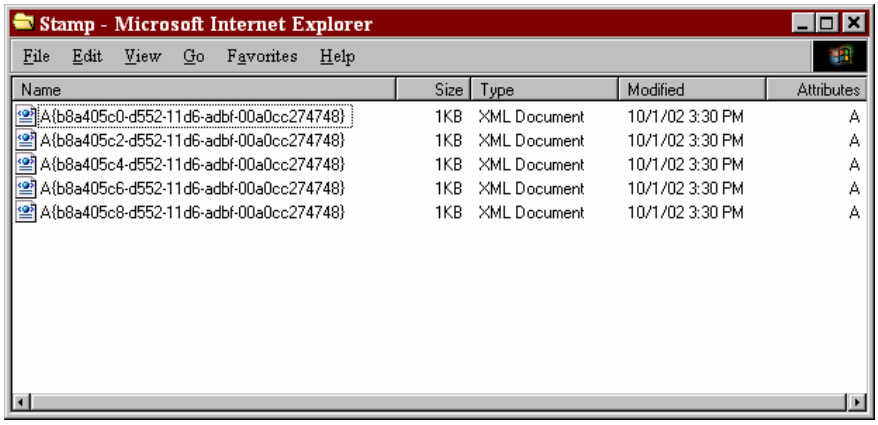
### Second Display

**Note** Pressing a key will terminate the program.

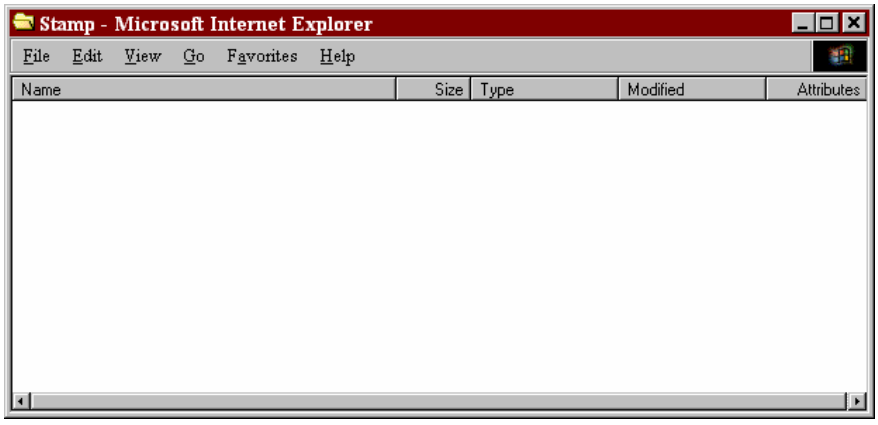
```
E:\xmlexample\Stamp\A{b8a405c0-d552-11d6-adbf-00a0cc274748}.xml imported by XMLImport
Contents: 15303258
 E:\xmlexample\Stamp\A{b8a405c2-d552-11d6-adbf-00a0cc274748}.xml imported by XMLImport
Contents: 15303264
 E:\xmlexample\Stamp\A{b8a405c4-d552-11d6-adbf-00a0cc274748}.xml imported by XMLImport
Contents: 15303264
 E:\xmlexample\Stamp\A{b8a405c6-d552-11d6-adbf-00a0cc274748}.xml imported by XMLImport
Contents: 15303264
 E:\xmlexample\Stamp\A{b8a405c8-d552-11d6-adbf-00a0cc274748}.xml imported by XMLImport
Contents: 15303264

You may now use IE to verify that the 'Stamp' directory has been emptied

Status: 0001
Informative:  1[0] - indicated directory contains no documents
Called from line 426 in EXAMPLE6(E:\xmlexample\EXAMPLE6.COB), compiled 2003/05/\
01 15:26:04.
E:\xmlexample\Stamp\*.xml
Press a key to terminate.
```

### XML Document

Microsoft Internet Explorer (or Windows Explorer) may be used to view the **Stamp** directory that contains the five generated XML documents. You can click on any document to see its content.




After continuing the program, the **Stamp** directory should empty out as shown.




# Example 7: Export File, Test Well-Formed File, and Validate File

This COBOL program illustrates how an XML document is generated from a COBOL data item and then how the syntax and content of an XML document may be verified.

The program first writes (or exports) an XML document file from the content of a COBOL data item. Then the program verifies that the generated document is well-formed. Finally, the program verifies that the content of the document conforms to the schema file that was generated by the **cobtoxml** utility.

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML TEST WELLFORMED-FILE (on page 69), which verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE FILE (on page 71), which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example7.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example7 y` |
| 2 | `cobtoxml example7 Liant-Address` |
| 3 | `move /y example7.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example7.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example7 k` |

Line 1 compiles the **example7.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 7 object filename is **example7.cob**,

and the model filenames are **example7.xml**, **example7.xtl**, **example7.xsl**, and **example7.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 7 object file, **example7.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example7.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application and opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **liant7.xml** using the XML EXPORT FILE statement.

Next, the syntax of **liant7.xml** is verified using the XML TEST WELLFORMED-FILE statement.

Following this, the content of **liant7.xml** is verified using the XML VALIDATE FILE statement.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

For the purposes of this example, both the XML TEST WELLFORMED-FILE and XML VALIDATE FILE statements were used. However, the XML VALIDATE FILE statement also tests an XML document for well-formed syntax.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "8911 Capital of Texas Highway North".
    02  Address-2  Pic X(64) Value "Suite 4300".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78759.
    02  Time-Stamp Pic 9(8).
```

This data item stores company address information (in this case, Liant's). The last field of the item is a time stamp containing the time that the program was executed. The reason for this item is to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example7.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT FILE`<br>`  Liant-Address`<br>`  "Liant7"`<br>`  "Example7".` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Verifying Syntax

| COBOL Statement | Description |
|---|---|
| `XML TEST WELLFORMED-FILE`<br>`  "Liant7".` | Execute the XML TEST WELLFORMED-FILE statement specifying the XML document filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Verifying Content

| COBOL Statement | Description |
|---|---|
| `XML VALIDATE FILE`<br>`  "Liant7"`<br>`  "Example7".` | Execute the XML VALIDATE FILE statement specifying:<br>the XML document filename<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `  Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `  Stop Run.` | Terminate the COBOL program. |
| `  Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named Z, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example 7

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### COBOL Display

**Note** Pressing a key will terminate the program.

Running the program (**runcobol example7**) produces the following display:

```
Example-7 - Illustrate TEST WELLFORMED-FILE and VALIDATE FILE
Liant7.xml exported by XML EXPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
11205270
Liant7.xml checked by XML TEST WELLFORMED-FILE
Liant7.xml validated by XML VALIDATE FILE

You may inspect 'Liant7.xml'

Status: 0000
Press a key to terminate:
```

### XML Document

Microsoft Internet Explorer may be used to view the generated XML document **liant7.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
  <name>Liant Software Corporation</name>
  <address-1>8911 Capital of Texas Highway North</address-1>
  <address-2>Suite 4300</address-2>
  <address-3>
    <city>Austin</city>
    <state>TX</state>
    <zip>78759</zip>
  </address-3>
  <time-stamp>11205270</time-stamp>
  </liant-address>
</root>
```

# Example 8: Export Text, Test Well-Formed Text, and Validate Text

This COBOL program illustrates how an XML document is generated from a COBOL data item and then how the syntax and content of an XML document may be verified. Next, the program verifies that the generated document is well-formed. Finally, the program verifies that the content of the document conforms to the schema file that was generated by the **cobtoxml** utility.

**Note** Under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT TEXT (on page 66), which constructs an XML document (as a text string) from the content of a COBOL data item.
- XML TEST WELLFORMED-TEXT (on page 70), which verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE TEXT (on page 72), which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML PUT TEXT (on page 74), which copies an XML document from a text string to a data file.
- XML FREE TEXT (on page 73), which releases the memory that was allocated by XML EXPORT TEXT to hold the XML document as a text string.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example8.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example8 y` |
| 2 | `cobtoxml example8 Liant-Address` |
| 3 | `move /y example8.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example8.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example8 k` |

Line 1 compiles the **example8.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 8 object filename is **example8.cob**, and the model filenames are **example8.xml**, **example8.xtl**, **example8.xsl**, and **example8.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 8 object file, **example8.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example8.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application and opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document as defined by the variable, *`Document-Pointer`*, using the XML EXPORT TEXT statement.

Next, the syntax of the generated XML document is verified using the XML TEST WELLFORMED-TEXT statement.

Following this, the content of the generated XML document is verified using the XML VALIDATE TEXT statement.

Next, the contents of the text string are written to a disk file using the XML PUT TEXT statement. The memory block is deallocated using the XML FREE TEXT statement. The primary aim of using the XML PUT TEXT statement is to make the content of the XML document available as an external file for viewing.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

For the purposes of this example, both the XML TEST WELLFORMED-TEXT and XML VALIDATE TEXT statements were used. However, the XML VALIDATE TEXT statement also tests an XML document for well-formed syntax.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "8911 Capital of Texas Highway North".
    02  Address-2  Pic X(64) Value "Suite 4300".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78759.
    02  Time-Stamp Pic 9(8).
```

This data item stores company address information (in this case, Liant's). The last field of the item is a time stamp containing the time that the program was executed. The reason for this item is to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT TEXT statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example8.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT TEXT`<br>`  Liant-Address`<br>`  "Document-Pointer"`<br>`  "Example8".` | Execute the XML EXPORT TEXT statement specifying:<br>the data item address,<br>the XML document text name,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Verifying Syntax

| COBOL Statement | Description |
|---|---|
| `XML TEST WELLFORMED-TEXT`<br>`  "Document-Pointer".` | Execute the XML TEST WELLFORMED-TEXT statement specifying the XML document text name. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Verifying Content

| COBOL Statement | Description |
|---|---|
| `XML VALIDATE TEXT`<br>`  "Document-Pointer"`<br>`  "Example8".` | Execute the XML VALIDATE TEXT statement specifying:<br>the XML document text name<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Copying an XML Document to a File

| COBOL Statement | Description |
|---|---|
| `XML PUT TEXT "Document-Pointer" "Liant8".` | Execute the XML PUT TEXT statement specifying: the XML document text name and the document filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Releasing the XML Document Memory

| COBOL Statement | Description |
|---|---|
| `XML FREE TEXT "Document-Pointer".` | Execute the XML FREE TEXT statement specifying the XML document text name. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the "Termination Test Logic" table). |
| `Stop Run.` | Terminate the COBOL program. |
| `Copy "lixmldsp.cpy".` | Copy in the status display logic (see the "Status Display Logic" table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named Z, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example 8

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

**Note** Pressing a key will terminate the program.

Running the program (**runcobol example8**) produces the following display:

```
Example-8 - Illustrate TEST-WELLFORMED TEXT and VALIDATE TEXT
Document exported by XML EXPORT TEXT
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
12545201
Document checked by XML TEST WELLFORMED-TEXT
Document validated by XML VALIDATE TEXT
Document memory written by XML PUT TEXT
Document memory released by XML FREE TEXT

You may inspect 'Liant8.xml'

Status: 0000
Press a key to terminate:
```

### XML Document

Microsoft Internet Explorer may be used to view the generated XML document, **liant8.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
    <name>Liant Software Corporation</name>
    <address-1>8911 Capital of Texas Highway North</address-1>
    <address-2>Suite 4300</address-2>
    <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78759</zip>
    </address-3>
    <time-stamp>12545201</time-stamp>
  </liant-address>
</root>
```

# Example 9: Export File, Transform File, and Import File

This COBOL program illustrates how an XML document is generated from a COBOL data item, and then how the content of an XML document may be converted into COBOL data format and stored in a COBOL data item.

The program first writes (or exports) an XML document file from the content of a COBOL data item. Next, the document is transformed into another format (the same format as described in Example 2: Export File and Import File with XSLT Stylesheets (on page 94) and then transformed back into the original output format. Then the program reads (or imports) the same XML document and places the content in the same COBOL data item. One additional transform is applied to add in the COBOL attributes to the input document.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE (on page 67), which reads an XML document (from a file) into a COBOL data item.
- XML TRANSFORM FILE (on page 70), which uses an XSLT stylesheet to modify (transform) an XML document into another format.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **example9.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol example9 y` |
| 2 | `cobtoxml example9 Liant-Address` |
| 3 | `move /y example9.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,example9.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol example9 k` |

Line 1 compiles the **example9.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example 9 object filename is **example9.cob**, and the model filenames are **example9.xml**, **example9.xtl**, **example9.xsl**, and **example9.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example 9 object file, **example9.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example9.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application and opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted,

line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **liant9a.xml** using the XML EXPORT FILE statement.

Next, the content of the XML document is transformed from the format that was used in Example 2 with an XML TRANSFORM FILE statement producing the file, **Liant9b.xml**, and then transformed back into the original output format.

Next, the content of the XML document is imported from the file, **liant9c.xml**, and placed in the same data item using the XML IMPORT FILE statement.

Subsequently, the content of the XML document, **liant9c.xml**, is transformed using the internal XSLT stylesheet from the set of model files creating the file, **liant9d.xml**. This adds all of the COBOL attributes to **liant9d.xml**.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "8911 Capital of Texas Highway North".
    02  Address-2  Pic X(64) Value "Suite 4300".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78759.
    02  Time-Stamp Pic 9(8).
```

This data item stores company address information (in this case, Liant's). The last field of the item is a time stamp containing the time that the program was executed. The reason for this item is to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example9.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT FILE`<br>`  Liant-Address`<br>`  "Liant9a"`<br>`  "Example9".` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Transforming to External XML Format

| COBOL Statement | Description |
|---|---|
| `XML TRANSFORM FILE`<br>`"Liant9a"`<br>`"toExt"`<br>`"Liant9b".` | Execute the XML TRANSFORM FILE statement specifying:<br>the input XML document filename,<br>the XSLT stylesheet filename,<br>and the output XML document filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Transforming to Internal XML Format

| COBOL Statement | Description |
|---|---|
| `XML TRANSFORM FILE`<br>`"Liant9b"`<br>`"toInt"`<br>`"Liant9c".` | Execute the XML TRANSFORM FILE statement specifying:<br>the input XML document filename,<br>the XSLT stylesheet filename,<br>and the output XML document filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Importing an XML Document

| COBOL Statement | Description |
|---|---|
| `Move Spaces to Liant-Address.` | Ensure that the Liant-Address item contains no data. |
| `XML IMPORT FILE`<br>`Liant-Address`<br>`"Liant9c"`<br>`"Example9".` | Execute the XML IMPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Transforming to Include COBOL Attributes

| COBOL Statement | Description |
|---|---|
| `XML TRANSFORM FILE`<br>`"Liant9c"`<br>`"Example9"`<br>`"Liant9df".` | Execute the XML TRANSFORM FILE statement specifying:<br>the input XML document filename,<br>the XSLT stylesheet filename,<br>and the output XML document filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the "Termination Test Logic" table). |
| `Stop Run.` | Terminate the COBOL program. |
| `Copy "lixmldsp.cpy".` | Copy in the status display logic (see the "Status Display Logic" table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph names `Z`, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

# Execution Results for Example 9

The following sections display the output of the COBOL program that is run and the XML document that is generated.

## *COBOL Display*

**Note** Pressing a key will terminate the program.

Running the program (**runcobol example9**) produces the following display:

```
Example-9 - Illustrate TRANSFORM FILE
Liant9a.xml exported by XML EXPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
14103001
Liant9a.xml transformed into Liant9b.xml by XML TRANSFORM FILE
Liant9b.xml transformed into Liant9c.xml by XML TRANSFORM FILE
Liant9c.xml imported by XML IMPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
14103001
Liant9c.xml transformed into Liant9d.xml by XML TRANSFORM FILE

You may inspect 'Liant9a.xml' - 'Liant9d.xml'

Status: 0000
Press a key to terminate:
```

## *XML Documents*

Microsoft Internet Explorer may be used to view the generated XML documents, **liant9a.xml**, **liant9b.xml**, **liant9c.xml**, and **liant9d.xml**. Their content of these documents should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

### Liant9a.xml – Internal Format (similar to Liant1.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
    <name>Liant Software Corporation</name>
    <address-1>8911 Capital of Texas Highway North</address-1>
    <address-2>Suite 4300</address-2>
    <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78759</zip>
    </address-3>
    <time-stamp>14103001</time-stamp>
  </liant-address>
</root>
```

### Liant9b.xml – External Format (similar to Liant2.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
<LiantAddress>
  <Information Name="Liant Software Corporation"
   Address1="8911 Capital of Texas Highway North"
   Address2="Suite 4300" City="Austin" State="TX" Zip="78759" />
  <TimeStamp Value="14103001" />
</LiantAddress>
```

### Liant9c.xml – Internal Format Restored

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
    <name>Liant Software Corporation</name>
    <address-1>8911 Capital of Texas Highway North</address-1>
    <address-2>Suite 4300</address-2>
    <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78759</zip>
    </address-3>
    <time-stamp>14103001</time-stamp>
  </liant-address>
</root>
```

### Liant9d.xml – Internal Format plus COBOL Attributes

```
<?xml version="1.0" encoding="UTF-8" ?>
<root type="nonnumeric" kind="GRP" dateTime="2003-05-14T14:10:24">
  <liant-address type="nonnumeric" kind="GRP" length="239" offset="4"
   id="Q1568">
    <name type="nonnumeric" kind="ANS" length="64" offset="4"
     id="Q1590">Liant Software Corporation</name>
    <address-1 type="nonnumeric" kind="ANS" length="64" offset="68"
     id="Q1612">8911 Capital of Texas Highway North</address-1>
    <address-2 type="nonnumeric" kind="ANS" length="64" offset="132"
     id="Q1634">Suite 4300</address-2>
    <address-3 type="nonnumeric" kind="GRP" length="39" offset="196"
     id="Q1656">
      <city type="nonnumeric" kind="ANS" length="32" offset="196"
       id="Q1678">Austin</city>
      <state type="nonnumeric" kind="ANS" length="2" offset="228"
       id="Q1700">TX</state>
      <zip type="numeric" kind="NSU" length="5" offset="230" scale="0"
       precision="5" id="Q1722">78759</zip>
    </address-3>
    <time-stamp type="numeric" kind="NSU" length="8" offset="235" scale="0"
     precision="8" id="Q1744">14103001</time-stamp>
  </liant-address>
</root>
```

# Example A: Diagnostic Messages

This program illustrates the diagnostic messages that may be displayed for XML documents that are not well-formed or valid. The program uses the XML TEST WELLFORMED-FILE and XML VALIDATE FILE statements to test and validate a series of XML documents. (These predefined XML documents are detailed in the Program Description section on page 154.)

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML TEST WELLFORMED-FILE (on page 69), which verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE FILE (on page 71), which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **exampleA.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol exampleA y` |
| 2 | `cobtoxml exampleA Liant-Address` |
| 3 | `move /y exampleA.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,exampleA.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol exampleA k` |

Line 1 compiles the **exampleA.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example A object filename is **exampleA.cob**, and the model filenames are **exampleA.xml**, **exampleA.xtl**, **exampleA.xsl**, and **exampleA.xsd**).

Lines 3, 4, and 5 are optional. They strip the symbol table from the example A object file, **exampleA.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **exampleA.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application and opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

In this example, three different predefined XML documents are processed:

- The **XLiantA1.xml** file is not well-formed and will cause the XML TEST WELLFORMED-FILE statement to return with an error status. Since this function fails, the XML VALIDATE FILE statement is not used to process this file.
- The **XLiantA2.xml** file is well-formed but not valid. The XML TEST WELLFORMED-FILE statement will return success. The XML VALIDATE FILE statement will return with an error status.
- The **XLiantA3.xml** file is both well-formed and valid. Both the XML TEST-WELLFORMED-FILE statement and the XML VALIDATE FILE statement will return a successful status.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "8911 Capital of Texas Highway North".
    02  Address-2  Pic X(64) Value "Suite 4300".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78759.
    02  Time-Stamp Pic 9(8).
```

This data item stores company address information (in this case, Liant's). The last field of the item is a time stamp containing the time that the program was executed. The reason for this item is to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

# Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **exampleA.cbl**.

## Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Testing for a Well-Formed Document

| COBOL Statement | Description |
|---|---|
| `XML TEST WELLFORMED-FILE`<br>`"Xliant1".` | Execute the XML TEST WELLFORMED-FILE statement specifying the XML document filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Testing for a Valid Document

| COBOL Statement | Description |
|---|---|
| `XML VALIDATE FILE`<br>`"XLiantA2"`<br>`"ExampleA".` | Execute the XML VALIDATE FILE statement specifying:<br>the XML document filename<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

## Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `Stop Run.` | Terminate the COBOL program. |
| `Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named Z, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example A

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

Running the program (**runcobol exampleA**) produces three displays: the first is shown after the first diagnostic message, the second is shown after the second diagnostic message, and the third is displayed after some successful tests.

**Note** Because of differences in the underlying XML parsers, the results of running Example A vary between Windows and UNIX. When a parser error occurs, the current UNIX implementation does *not* display the offending line of XML text in

error (as shown in the first display). In addition, because the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library under UNIX, errors that would be detected by a schema are not reported (as illustrated in the second display). The third display, however, is the same under both implementations.

### First Display

**Note** Pressing a key will cause the program to continue.

For Windows, the first display would be illustrated as:

```
Example-A - Illustrate diagnostics for invalid documents
 and documents that are not well-formed
XML TEST WELLFORMED-FILE - not well-formed
Error: 28[10] - in function: LoadDocument
Called from line 398 in EXAMPLEA(E:\xmlexample\EXAMPLEA.COB), compiled 2003/05/\
   08 13:05:56.
E:\xmlexample\XLiantA1.xml
End tag 'rm-address' does not match the start tag 'liant-address'.
line 2, position 262
<root><liant-address><name>Liant Software Corporation</name><address-1>8911 Cap\
ital of Texas Highway North</address-1><address-2>Suite 4300</address-2><address\
s-3><city>Austin</city><state>TX</state><zip>78759</zip></address-3><time-stamp\
>14525751</time-stamp></rm-address></root>
------------------------------------------------------------------------------\
------------------------------------------------------------------------------\
------------------------------------------------------------------------------\
-----------------------|
Press a key to continue:
```

For UNIX, the first display would be shown as follows:

```
Example-A - Illustrate diagnostics for invalid documents
 and documents that are not well-formed
XML TEST WELLFORMED-FILE - not well-formed
Error: 28[2] - in function: LoadDocument
Called from line 431 in EXAMPLEA(/usr/xmltk/examples/examplea.cob), compiled\
   2003/05/14 13:27:55.
xliantal.xml
Press a key to continue:
```

### Second Display

**Note** Pressing a key will cause the program to continue.

For Windows, the second display would be illustrated as:

```
XML TEST WELLFORMED-FILE - well-formed – invalid
XML VALIDATE FILE - well-formed – invalid
Error: 28[10] - in function: LoadDocument
Called from line 411 in EXAMPLEA(E:\xmlexample\EXAMPLEA.COB), compiled 2003/05/\
08 13:05:56.
E:\xmlexample\XLiantA2.xml
The value of 'ABCDE' is invalid according to its data type.  The element: 'zip'\
  has an invalid value according to its data type.
line 2, position 211
<root><liant-address><name>Liant Software Corporation</name><address-1>8911 Cap\
ital of Texas Highway North</address-1><address-2>Suite 4300</address-2><address\
s-3><city>Austin</city><state>TX</state><zip>ABCDE</zip></address-3><time-stamp\
>14525751</time-stamp></liant-address></root>
--------------------------------------------------------------------------------\
--------------------------------------------------------------------------------\
-------------------------------------------------|
Press a key to continue:
```

For UNIX, the second display would be shown as follows:

```
XML TEST WELLFORMED-FILE - well-formed – invalid
XML VALIDATE FILE - well-formed – invalid
Press a key to continue:
```

### Third Display

**Note** Pressing a key will terminate the program.

```
XML TEST WELLFORMED-FILE - well-formed – valid
XML VALIDATE FILE - well-formed – valid
Status: 0000
Press a key to terminate:
```

# Example B: Import File with Missing Intermediate Parent Names

This COBOL program illustrates how an XML document with some missing intermediate parent names may be converted into COBOL data format and stored in a COBOL data item. (This capability of handling missing intermediate parent names has been included to make programs that deal with "flattened" data items, such as Web services, less complicated.) A COBOL program and an XML document file may contain the same elementary items, but may not have the identical structure. XML Extensions offers a way to handle such cases where there is not a one-to-one match between the COBOL data item and the XML document structure. Consider the following situation, in which the COBOL program imports a predefined XML document that has some missing intermediate parent names.

A missing intermediate parent name is an XML element name that corresponds to an intermediate level COBOL group name. For example, in the following COBOL data item, the XML element name, `address-3`, is an intermediate parent name.

```
01  MY-ADDRESS.
    02  ADDRESS-1        PIC X(64) VALUE "101 Main St.".
    02  ADDRESS-2        PIC X(64) VALUE "Apt 2B".
    02  ADDRESS-3.
        03  CITY         PIC X(32) VALUE "Smallville".
        03  STATE        PIC X(2)  VALUE "KS".
```

The structure of the corresponding XML document would be:

```
<root>
  <my-address>
    <address-1>101 Main St.</address-1>
    <address-2>Apt 2B</address-2>
    <address-3>
      <city>Smallville</city>
      <state>KS</state>
    </address-3>
  <my-address>
<root>
```

In cases where the intermediate parent name is not needed to resolve ambiguity, XML Extensions will attempt to reconstruct the document structure on input. For example, if the input XML document contained the following information, then the intermediate parent names of `address-3` and `my-address` would be added to produce an XML document compatible with the above document.

```
<root>
  <address-1>101 Main St.</address-1>
  <address-2>Apt 2B</address-2>
  <city>Smallville</city>
  <state>KS</state>
<root>
```

Example B illustrates this situation more fully.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 67), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **exampleB.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol exampleB y` |
| 2 | `cobtoxml exampleB Liant-Address -sn` |
| 3 | `move /y exampleB.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,exampleB.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `start /w runcobol exampleB k` |

Line 1 compiles the **exampleB.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example B object filename is **exampleB.cob**, and the model filenames are **exampleB.xml**, **exampleB.xtl**, and **exampleB.xsl**). The **-sn** (schema none) option on the **cobtoxml** utility disables the

generation of a schema file, which is normally used to validate the content of an XML document. Note that under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

Lines 3, 4, and 5 are optional. They strip the symbol table from the example B object file, **exampleB.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **exampleB.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application and opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

This COBOL program illustrates how an XML document with some missing intermediate parent names may be converted into COBOL data format and stored in a COBOL data item.

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **LiantB.xml** using the XML EXPORT FILE statement.

Next, the content of the XML document is imported from the file, **LiantB.xml**, and placed in the same data item using the XML IMPORT FILE statement.

Additionally, the content of the predefined XML document named **XLiantB.xml**, which has some missing intermediate parent names, is also imported using the XML IMPORT FILE statement.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item defined in the copy file, **liant.cpy**, is as follows:

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "8911 Capital of Texas Highway North".
    02  Address-2  Pic X(64) Value "Suite 4300".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78759.
    02  Time-Stamp Pic 9(8).
```

This data item stores company address information (in this case, Liant's). The last field of the item is a time stamp containing the time that the program was executed. The reason for this item is to assure the person observing the execution of the example that the results are current. The time element in the generated XML document should change each time the example is run and should contain the current time.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **exampleB.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `Accept Time-Stamp From Time.` | Populate the Time-Stamp field. |
| `XML EXPORT FILE`<br>`  Liant-Address`<br>`  "LiantB"`<br>`  "ExampleB".` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Importing an XML Document

| COBOL Statement | Description |
|---|---|
| `Move Spaces to Liant-Address.` | Ensure that the Liant-Address item contains no data. |
| `XML IMPORT FILE`<br>`  Liant-Address`<br>`  "LiantB"`<br>`  "ExampleB".` | Execute the XML IMPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>and the model filename. |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `Stop Run.` | Terminate the COBOL program. |
| `Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named Z, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example B

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

**Note** Pressing a key will terminate the program.

Running the program (**runcobol exampleB**) produces the following display:

```
Example-B - Illustrate IMPORT with missing intermediate names
LiantB.xml exported by XML EXPORT FILE
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
16480895
LiantB.xml imported by XML IMPORT FILE:
Liant Software Corporation
8911 Capital of Texas Highway North
Suite 4300
Austin                          TX78759
16480895
XLiantB.xml imported by XML IMPORT FILE:
Wild Hair Corporation
8911 Hair Court
Sweet 4300
Lostin                          TX70707
99999999
You may inspect 'LiantB.xml' and 'XLiantB.xml'

Status: 0000
Press a key to terminate:
```

## *XML Document*

Microsoft Internet Explorer may be used to view the generated XML document, **LiantB.xml** and the predefined XML document **XLiantB.xml**. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

### LiantB.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <liant-address>
    <name>Liant Software Corporation</name>
    <address-1>8911 Capital of Texas Highway North</address-1>
    <address-2>Suite 4300</address-2>
    <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78759</zip>
    </address-3>
    <time-stamp>16480895</time-stamp>
  </liant-address>
</root>
```

### XLiantB.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <name>Wild Hair Corporation</name>
  <address-1>8911 Hair Court</address-1>
  <address-2>Sweet 4300</address-2>
  <city>Lostin</city>
  <state>TX</state>
  <zip>70707</zip>
  <time-stamp>0</time-stamp>
</root>
```

# Example C: Export File with Document Prefix

This program writes (or exports) an XML document file from the content of a COBOL data item. A document prefix is specified to declare entities referenced in the COBOL data item.

This example uses the following XML statements:

- XML INITIALIZE (on page 79), which initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (on page 64), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML TERMINATE (on page 80), which terminates or closes the session with the **xmlif** library.

**Note** The XML EXPORT FILE statement contains an additional parameter: the name of the document prefix being used for the XML document export.

## Development

The COBOL program must be compiled with the RM/COBOL compiler, and the output symbol table option (Y Compile Command Option) must be enabled.

After each successful compilation, it is necessary to run the **cobtoxml** utility program to generate a set of model files that are used by the XML IMPORT and XML EXPORT statements.

After the successful execution of the **cobtoxml** utility, you may then execute the COBOL program. The **xmlif** library may be specified either by entering it on the command line (for example, **runcobol myprog l="some\path\xmlif"**) or by placing the **xmlif** library in the **rmautold** directory (this is normally a subdirectory of the RM/COBOL installation directory).

Once the program is tested, you may choose to delete the symbol table information from the COBOL object file by using the RM/COBOL Combine Program (**rmpgmcom**) utility.

## Batch File

The following DOS commands may be entered into a batch file. These commands build and execute **exampleC.cob**.

| Line | Statement |
|---|---|
| 1 | `rmcobol exampleC y` |
| 2 | `cobtoxml exampleC Line-Item -sn` |
| 3 | `move /y exampleC.cob tmp.cob` |
| 4 | `start /w runcobol rmpgmcom A='STRIP,exampleC.cob,tmp.cob'` |
| 5 | `del tmp.cob` |
| 6 | `Start /w runcobol exampleC k` |

Line 1 compiles the **exampleC.cbl** source file with the symbol table option (Y) enabled.

Line 2 builds the XML model files from the symbol table information in the symbol table. By default, the model filenames are the same as the object filename with different extensions (in this instance, the example C object filename is **exampleC.cob**, and the model filenames are **exampleC.xml**, **exampleC.xtl**, and **exampleC.xsl**). The **-sn** (schema none) option on the **cobtoxml** utility disables the generation of a schema file, which is normally used to validate the content of an XML document. Note that under UNIX, the schema file that is produced by the **cobtoxml** utility is ignored by the **xmlif** library.

Lines 3, 4, and 5 are optional. They strip the symbol table from the example C object file, **exampleC.cob**. In order to reduce the size of the deployed object files, developers may chose to remove the symbol table from the COBOL object file before distributing their applications. The RM/COBOL Combine Program (**rmpgmcom**) utility, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **exampleC.cob**. The K Option suppresses the runtime banner. On line 6, the `start /w` sequence is included only as good programming practice.

**Note** On Windows, the RM/COBOL runtime (**runcobol**) is a Windows application that opens a separate window when executed from DOS. The `start /w` part of the DOS command instructs Windows to start the runtime and then wait (the W Option) for its completion. This step is necessary in line 4. If this step were omitted, line 5 could execute before the runtime completed, which would cause the input file (**tmp.cob**) passed to **rmpgmcom** to be deleted before it had been completely read.

## Program Description

This COBOL program illustrates how an XML document is generated from a COBOL data item, and then how the content of an XML document may be converted into COBOL data format and stored in a COBOL data item.

Before any other XML statement may be executed, the XML INITIALIZE statement must be successfully executed. Since it is possible for XML INITIALIZE to fail, the return status must be checked before continuing.

Data is exported from the data item `Line-Item` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **liantC.xml** using the XML EXPORT FILE statement.

Finally, the XML interface is terminated with the XML TERMINATE statement.

If any of the statements terminate unsuccessfully, the XML GET STATUS-TEXT statement is called.

## Data Item

The content of the COBOL data item `Line-Item` is as follows:

```
01  Line-Item.
    02 LI-BoilerPlate.
       03 FILLER      VALUE "&BoilerPlate;".
    02 LI-Name        PIC X(30).
    02 LI-Quantity    PIC 9(04).
    02 LI-CurPrice.
       03 FILLER      VALUE "&CURRENCY;".
       03 LI-Price    PIC 9(06)V99.
    02 LI-CurExt.
       03 FILLER      VALUE "&CURRENCY;".
       03 LI-Ext      PIC 9(10)V99.
```

This data item stores line item information, such as might appear in an invoice. There are three entity references, one to BoilerPlate and two to CURRENCY.

## Document Prefix

The document prefix string is as follows:

```
78  DocumentPrefix    Value
      "<!DOCTYPE root [" & LF &
      "   <!ENTITY CURRENCY ""&#036;"">" & LF &
      "   <!ENTITY BoilerPlate ""All prices in USD"">" & LF &
      "       ]>".
```

This document prefix string provides a document type definition (DTD) that declares two entities in the internal subset. The first entity is CURRENCY, which is defined to be #036, the "$" in UTF-8. The second entity is BoilerPlate, which is defined to be "All prices in USD". The string will produce multiple lines in the exported document file because of the line feed characters introduced by the symbolic-character name LF.

## Other Definitions

The copy file, **lixmlall.cpy**, should be included in the Working-Storage Section of the program.

The copy file, **lixmldef.cpy**, which is copied in by **lixmlall.cpy**, defines a data item named `XML-data-group`. The content of this COBOL data item is as follows:

```
01  XML-data-group.
    03  XML-Status             PIC 9(4).
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-Success
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE XML-InformDirectoryEmpty.
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC 9 BINARY(1).
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC 9(10) BINARY(4).
```

Various XML statements may access one of more fields of this data item. For example, the XML EXPORT FILE statement returns a value in the `XML-Status` field. The XML GET STATUS-TEXT statement accesses the `XML-StatusText` and `XML-MoreFlag` fields.

## Program Structure

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **exampleC.cbl**.

### Initialization

| COBOL Statement | Description |
|---|---|
| `XML INITIALIZE.` | Execute the XML INITIALIZE statement (no parameters). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Exporting an XML Document

| COBOL Statement | Description |
|---|---|
| `XML EXPORT FILE`<br>`  Line-Item`<br>`  "LiantC"`<br>`  "ExampleC"`<br>`  OMITTED  *> no stylesheet`<br>`  DocumentPrefix.` | Execute the XML EXPORT FILE statement specifying:<br>the data item address,<br>the XML document filename,<br>the model filename,<br>the XSLT stylesheet (OMITTED),<br>and the XML document prefix (DTD). |
| `If Not XML-OK Go To Z.` | If the statement terminates unsuccessfully, go to the termination logic. |

### Program Exit Logic

| COBOL Statement | Description |
|---|---|
| `Z.` | Paragraph-name that is a target of error condition GO TO statements. |
| `  Copy "lixmltrm.cpy".` | Copy in the termination test logic (see the “Termination Test Logic” table). |
| `  Stop Run.` | Terminate the COBOL program. |
| `  Copy "lixmldsp.cpy".` | Copy in the status display logic (see the “Status Display Logic” table). |

### Termination Test Logic

This code is found in the copy file, **lixmltrm.cpy**.

This code occurs after the paragraph named Z, so that any error condition is obtained here via a GO TO Z statement. If there are no errors, execution "falls through" to these statements.

| COBOL Statement | Description |
|---|---|
| `Display "Status: " XML-Status.` | Display the most recent return status value (if there are no errors, this should display zero). |
| `Perform Display-Status.` | Perform the Display-Status paragraph to display any error messages. |
| `XML TERMINATE.` | Terminate the XML interface. |
| `Perform Display-Status.` | Perform the Display-Status paragraph again to display any error encountered by the XML TERMINATE statement. |

### Status Display Logic

This code is found in the copy file, **lixmldsp.cpy**.

This code is called twice by the termination test logic: the first time to report any error condition that exists, and the second time to report an error (if one occurs) from the XML TERMINATE statement. If there are no errors (the condition `XML-IsSuccess` is true), this paragraph displays no information.

| COBOL Statement | Description |
|---|---|
| `Display-Status.` | This is the paragraph-name. |
| `If Not XML-IsSuccess` | Do nothing if XML-IsSuccess is true. |
| `Perform With Test After Until XML-NoMore` | Perform as long as there are status lines available to be displayed (until XML-NoMore is true). |
| `XML GET STATUS-TEXT` | Get the next line of status information from the XML interface. |
| `Display XML-StatusText` | Display the line that was just obtained. |
| `End-Perform` | End of the perform loop. |
| `End-If.` | End of the IF statement and the paragraph. |

## Execution Results for Example C

The following sections display the output of the COBOL program that is run and the XML document that is generated.

### *COBOL Display*

**Note** Pressing a key will terminate the program.

Running the program (**runcobol exampleC**) produces the following display:

```
Example-C - Illustrate EXPORT FILE with Document Prefix
LiantC.xml exported by XML EXPORT FILE

You may inspect 'LiantC.xml'

Status: 0000
Press a key to terminate:
```

### *XML Document*

Microsoft Internet Explorer may be used to view the generated XML document, **liantC.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE root (View Source for full doctype...)>
<root>
    <line-item>
       <li-boilerplate>All prices in USD</li-boilerplate>
       <li-name>Widget</li-name>
       <li-quantity>50</li-quantity>
       <li-curprice>
           $
           <li-price>5.42</li-price>
       </li-curprice>
       <li-curext>
           $
           <li-ext>271.00</li-ext>
       </li-curext>
    </line-item>
</root>
```

In this display, the BoilerPlate entity reference is replaced with the DTD declared value “All prices in USD” and the CURRENCY entity references are replaced with the DTD declared value “$” (#036 in UTF-8). If you view the source, the original entity references will be shown instead.

# Example Batch Files

Three batch files are provided to facilitate use of the example programs: **cleanup.bat**, **example.bat**, and **examples.bat**.

## Cleanup.bat

This batch file will remove various files that were created by executing the example programs. This file contains a series of delete file commands similar to the following:

```
@echo off
@echo cleanup
if exist liant*.xml del liant*.xml
if exist table1.xml del table1.xml
if exist table2.xml del table2.xml
if exist table3.xml del table3.xml
if exist table4.xml del table4.xml
if exist example*.cob del example*.cob
if exist tmp.cob del tmp.cob
if exist *.lst del *.lst
if exist example*.x* del example*.x*
if exist Stamp\*.xml del Stamp\*.xml
if exist Stamp rmdir Stamp
```

This batch file has no parameters. Run it by entering the following on the command line:

```
Cleanup
```

## Example.bat

This batch file will compile a COBOL source program, run the **cobtoxml** utility against the compiled object code, delete the symbol table from the object code, and finally execute the COBOL program. The content of this file is as follows:

```
         rmcobol  %1 y k
         cobtoxml %1 %2 %3 –bn
         if exist tmp.cob del tmp.cob
         rename   %1.cob tmp.cob
start /w runcobol rmpgmcom A='STRIP,%1.cob,tmp.cob'
start /w runcobol %1 k
```

This batch file uses parameters that are specified by the caller of the batch file. The first parameter is the filename of the COBOL program (without the **.cbl** extension). The second parameter is the name of a data-item within the COBOL program, from which the **cobtoxml** utility will construct model files. The third parameter is used for passing options to the **cobtoxml** utility.

To build and run Example 1: Export File and Import File (on page 88) using this batch file, enter the following on the command line:

```
example Example1 Liant-Address
```

## Examples.bat

This batch file will clean up files that were created from a previous run and then compile and run each example. The content of this file is similar to the following:

```
@echo off
call cleanup

@echo Example1 - Export / Import File.
call example example1 Liant-Address

@echo Example2 - Export / Import with XSLT stylesheets.
call example example2 Liant-Address

@echo Example3 - Export / Import with Occurs Depending.
call example example3 Liant-Address

@echo Example4 - Export / Import with sparse arrays.
call example example4 Data-Table –sn

@echo Example5 - Export / Import Text.
call example example5 Liant-Address

@echo Example6 - Export / Import with directory polling.
mkdir Stamp
call example example6 Time-Stamp

@echo Example7 - Export / Well-Formed File / Validate File.
call example example7 Liant-Address

@echo Example8 - Export / Well-Formed Text / Validate Text.
call example example8 Liant-Address

@echo Example9 - Export / Transform / Import.
call example example9 Liant-Address

@echo ExampleA - Well-Formed / Validate diagnostics.
call example exampleA Liant-Address

@echo ExampleB - Import with missing intermediate names.
call example exampleB Liant-Address -sn

@echo ExampleC - Export with document prefix.
call example exampleC Line-Item -sn
```

This batch file has no parameters. Run it by entering the following on the command line:

```
Examples
```

# Appendix B: XML Extensions Sample Application Programs

XML Extensions for RM/COBOL provides several complete and useful sample application programs. The purpose of these self-contained programs is to demonstrate and explain how to perform typical application-building tasks in XML Extensions within a realistic context so that you can better see how to integrate them into your own applications.

## Accessing the Sample Application Programs

The sample application programs are included in the XML Extensions samples directory, **Samples**.

Each sample application program is intended to reside in a separate subdirectory. For example, the XFORM sample resides in the directory named **Samples/xform**. Documentation for the sample is contained in the directory in the form of an HTML file.

On Windows systems, each application is packaged as a self-extracting executable program. For example, the XFORM sample is contained in the file **Samples/xform/xform.exe**. Running this application will extract its component parts. For the XFORM sample, this will produce the files, **xform.cbl** and **xform.htm**.

On UNIX systems, the applications were extracted when the samples were installed. The XFORM sample in **Samples/xform** contains the files, **xform.cbl** and **xform.htm**.

# Appendix C: XML Extensions Error Messages

This appendix lists and describes the messages that can be generated during the use of XML Extensions for RM/COBOL.

## Error Message Format

XML Extensions error messages may be several lines long. The general format of an error message includes the text of the message, and, if available, the COBOL traceback information, the name of the file or data item, and the parser information.

**Note** See Table 1 (on page 179) for a summary of error messages.

### Message Text

The first line of the error message has the following format:

```
<severity> - <message number> <message text>
```

*severity* indicates the gravity and type of message: Informative, Warning, or Error.

*message number* is the documented message number followed by an internal message number in bracket characters. The internal number provides information for Liant technical support to use in diagnosing problems.

*message text* is a brief explanation for the cause of the error.

An example of the first line of an error message is shown below:

```
Error: 28[12] - in function: LoadDocument
```

## COBOL Traceback Information

The second line of the error message, present if the information is available, contains COBOL traceback information such as the following:

```
Called from line 421 in TEST15.COB(C:\DEV\TEST15.COB),
compiled 2003/05/14 09:42:06.
```

The error-reporting facility will try to break up lines that are too long for the line buffer provided in the COBOL program. This prevents long lines from being truncated. A backward slash character (\) is placed in the last position of the buffer and the line is continued on the subsequent line. For example, the traceback line shown above may be broken up as follows:

```
Called from line 421 in TEST15.COB(C:\DEV\TEST15.COB), co\
mpiled 2003/05/14 09:42:06.
```

## Filename or Data Item in Error

The third line of the error message, present if the information is available, normally contains the name of the file or data item in error being referenced.

## Parser Information

**Note** This section applies to the Windows implementation of XML Extensions for RM/COBOL only.

Additional lines may be present that contain parser or schema diagnostics from the underlying XML parser, such as:

```
Error parsing 'a9' as number datatype.
line 5, position 16
<ItemCount>a9</ItemCount>
---------------|
```

The first line of parser or schema diagnostic information contains an error message. The second line contains the line number and column position within the XML document. The third line contains the line of XML text in error. The fourth line contains an indicator that draws attention to the column position.

# Summary of Error Messages

Table 1 describes the messages that may be generated when an error occurs in XML Extensions for RM/COBOL.

**Table 1: XML Extensions Error Messages**

| Message Number | Severity and Message Text | Description |
|---|---|---|
| 0 | Success | A normal completion occurred. No informative, warning, or error message was detected. |
| 1 | Informative - directory contains no documents | An XML FIND FILE statement did not find any XML documents (files with a .xml extension) in the specified directory. |
| 2 | Informative - document file - no data | An XML EXPORT FILE or XML EXPORT TEXT statement generated a document that contained no element values. |
| 3 | Warning - internal logic - memory not deallocated | During process cleanup, memory blocks that should have already been deallocated were still allocated. |
| 4 | Warning - invalid option - ignored | The cobtoxml utility has detected an invalid command line option. The option is ignored and processing continues. |
| 5 | Error - COBOL object file - invalid format | The cobtoxml utility has detected that the specified COBOL object file is not valid. This usually means that the header checksum is invalid. |
| 6 | Error - COBOL object file - open failure | The cobtoxml utility detected an error while attempting to open the specified COBOL object file. |
| 7 | Error - COBOL object file - read failure | The cobtoxml utility detected an error while attempting to read data from the specified COBOL object file. |
| 8 | Error - COBOL object file - seek failure | The cobtoxml utility detected an error while attempting to seek to a location within the specified COBOL object file. |
| 9 | Error - in function: CreateDocument | The underlying XML parser detected an error while trying to create an XML document. This error may occur in the cobtoxml utility or the xmlif library. |
| 10 | Error - cannot create URL | The xmlif library detected that a URL (a string beginning with the sequence “http://” or “https://”) was used as an output document name. |
| 11 | Error – data item – duplicate found | The cobtoxml utility has detected that there is more than one occurrence of the specified data item name in the COBOL object file or library. |

**Table1: XML Extensions Error Messages (*Cont.*)**

| Message Number | Severity and Message Text | Description |
|---|---|---|
| 12 | Error – data item – not found | The cobtoxml utility has detected that there are no occurrences of the specified data item name in the COBOL object file or library. |
| 13 | Error – document file – create failure | An attempt to create an XML document file has failed. This error may occur in the xmlif library or the cobtoxml utility. |
| 14 | Error – document file – file open failure | The xmlif library detected an error while attempting to open an XML document file. |
| 15 | Error – extraneous element | The xmlif library has detected an extra occurrence of a scalar data element. |
| 16 | Error – example file – create failure | The cobtoxml utility detected an error while attempting to create an example file. |
| 17 | Error – in function: GetFirstChild | The xmlif library detected an error in the function GetFirstChild while parsing an XML document. |
| 18 | Error – in function: GetNextSibling | The xmlif library detected an error in the function GetNextSibling while parsing an XML document. |
| 19 | Error – in function: GetNodeData | The xmlif library detected an error in the function GetNodeData while parsing an XML document. |
| 20 | Error – in function: GetRootNode | The xmlif library detected an error in the function GetRootNode while parsing an XML document. |
| 21 | Error – internal logic – memory allocation | An attempt to allocate a block of memory failed. This error may occur in either the cobtoxml utility or the xmlif library. |
| 22 | Error – internal logic – memory corruption | An attempt to deallocate (free) a block of memory failed either because the block header or trailer was corrupted or because the free memory call returned an error. This error may occur in either the cobtoxml utility or the xmlif library. |
| 23 | Error – internal logic – node not found | The xmlif library has detected an inconsistency in its internal tables. Specifically an expected entry in the Document Object Model is missing. |
| 24 | Error – in function: Initialization | Either an XML statement (other than XML INITIALIZE) was executed without first executing the XML INITIALIZE statement or the XML INITIALIZE statement failed. This error may occur in the xmlif library. In addition, improper installation of the underlying XML parser could cause the cobtoxml utility to fail with this error while attempting to generate an XSLT stylesheet or schema. |

**Table1: XML Extensions Error Messages (*Cont.*)**

| Message Number | Severity and Message Text | Description |
|---|---|---|
| 25 | Error – invalid data address | The xmlif library has detected that the data structure address specified in XML import or export statements does not match the data address specified in the template file. This normally means that the COBOL program has been re-compiled but that the cobtoxml utility was not re-executed to regenerate the model files. |
| 26 | Error – invalid object time stamp | The xmlif library while attempting to execute an XML import OR export statements has detected that the time stamp of the COBOL object used in generating the model files does not match the time stamp of the COBOL object being executed. This normally means that the COBOL program has been re-compiled but that the cobtoxml utility was not re-executed to regenerate the model files. |
| 27 | Error – license management | The license verification logic in the cobtoxml utility detected an error. |
| 28 | Error – in function: LoadDocument | An error was detected while trying to load an XML document. This normally means that there was a problem locating the document (either the document does not exist or there is a problem with permissions). This error may occur in either the xmlif library or the cobtoxml utility. Occasionally, XML Extensions generates documents that are then loaded as input documents. In the unlikely event that the generated document contains errors, a load document error will be encountered. |
| 29 | Error – in function: LoadSchema | An error was detected while trying to load an XML schema file. This normally means that there was a problem locating the document (either the document does not exist or there is a problem with permissions) or that the schema itself is in error. This error may occur in either the xmlif library or the cobtoxml utility. |
| 30 | Error - in function: LoadStyleSheet | An error was detected while trying to load an internal or external XSLT stylesheet. This normally means that there was a problem locating the document (either the document does not exist or there is a problem with permissions). Another possible cause is that the XML parser software is not properly installed. This error may occur in either the xmlif library or the cobtoxml utility. |

**Table1: XML Extensions Error Messages (*Cont.*)**

| Message Number | Severity and Message Text | Description |
|---|---|---|
| 31 | Error - in function: LoadStyleSheetFromText | An error was detected while trying to load an XSLT stylesheet. This normally means that there was a problem locating the document (either the document does not exist or there is a problem with permissions). Another possible cause is that the XML parser software is not properly installed. This error may occur in the cobtoxml utility. |
| 32 | Error - in function: LoadTemplate | An error was detected while trying to load an XML template file. This normally means that there was a problem locating the document (either the document does not exist or there is a problem with permissions). Another possible cause is that the XML parser software is not properly installed. This error may occur in the xmlif library. |
| 33 | Error - parameter - COBOL object file name missing | The cobtoxml utility has detected that the COBOL object file name command-line parameter is missing. |
| 34 | Error - parameter - data item name missing | The cobtoxml utility has detected that the name of the data item command-line parameter is missing. |
| 35 | Error - subscript out of range | The xmlif library, while executing an XML IMPORT FILE or XML IMPORT TEXT statement, has detected that a subscript reference is out of range (the subscript value is greater than the maximum for the array). This may occur either when the subscript is explicitly supplied in an attribute or when the subscript is generated implicitly (when an extra occurrence is present). |
| 36 | Error - temporary file access error | The xmlif library uncounted error while attempting to access a temporary intermediate file. This error can occur during the XML IMPORT TEXT, XML EXPORT TEXT, XML VALIDATE TEXT, or XML TEST WELLFORMED-TEXT statements. |
| 37 | Error - in function: TransformDOM | An unexpected error occurred while performing an XSLT transform of an XML document. This is most likely an internal error. This error may occur in either the xmlif library or the cobtoxml utility. |
| 38 | Error - in function: TransformText | An error occurred while performing an XSLT transform of an XML document using an external (user-supplied) XSLT stylesheet. This error may occur in the xmlif library. |

**Table1: XML Extensions Error Messages (*Cont.*)**

| Message Number | Severity and Message Text | Description |
|---|---|---|
| 39 | Error - symbol table - not found | This cobtoxml utility could not find the symbol table information in the COBOL object. This normally indicates that the COBOL program needs to be recompiled using the Y option. |
| 41 | Error - old runtime version | The cobtoxml utility has detected that the current COBOL runtime version is not supported. An RM/COBOL version 8 or later runtime system is required. |
| 42 | Error - in function: WriteDocument | An error occurred while attempting to write an XML document from the internal Document Object Model representation. This error may occur in either the xmlif library or the cobtoxml utility. |
| 43 | Error - wrong COBOL object version | The cobtoxml utility has determined that the COBOL object version being used is newer than was available when this version of XML Extensions was released and, therefore, may contain features that are not supported by XML Extensions. Check with Liant Software for updates to XML Extensions. |
| 44 | Error - wrong cobtoxml revision | The xmlif library has determined that the format of the model files may be incompatible with the xmlif library. This normally indicates that a new version of XML Extensions is being used but that the model files were generated with an older cobtoxml utility. |
| 45 | Error - invalid encoding selection | The value of the XML SET ENCODING parameter was neither “local” nor “utf-8”. |
| 46 | Error - invalid UTF-8 data | An XML export operation failed because the data supplied was not valid for UTF-8. |
| 47 | Error - invalid RM_ENCODING value | The value supplied in the RM_ENCODING environment variable was not recognized by the local iconv library. |
| 48 | Error - unable to locate iconv library | The xmlif library was unable to locate and open the iconv library. |
| 49 | Error - directory open failure | The XML FIND FILE statement was not able to locate and open the specified directory. |

# Appendix D: Summary of Enhancements

This appendix summarizes the enhancements from previous releases of XML Extensions, beginning with the most recent release.

**Note** Beginning with the version 9 release, the name of this product changed from XML Toolkit to XML Extensions.

## Version 2

This section summarizes the major enhancements available in version 2 of XML Toolkit:

- **Support for UNIX**. XML Extensions is currently available for selected UNIX platforms, including AIX, HP-UX, Linux, SCO OpenServer, Sun Solaris, and UnixWare.

  The Windows implementation continues to use Microsoft's XML parser, MSXML 4.0 or greater. The UNIX implementation is based on the XML parser (libxml) and the XSLT transformation parser (libxslt) from the C libraries for the Gnome project. While the Windows implementation continues to support the use of schema files, the UNIX implementation of schema support in the underlying XML parser (libxml) is still under development.

- Document Type Definition Support (on page 53). Exporting of XML documents has been enhanced to include the ability to specify a document type definition, which defines the legal building blocks of an XML document. A DTD can be used to define entity names that are referred to by the values of FILLER data items in the COBOL data structure being exported.

- Anonymous COBOL Data Structures (on page 48). The acts of exporting and importing of documents have been improved so that an anonymous COBOL data structure can be used. An anonymous COBOL data structure is any data area that is the same size or larger than the data structure indicated by the template file. This means that exporting from and importing to a Linkage Section data item, which is either based on an argument passed to a called program or a pointer set by the SET statement (for example, into allocated memory), is now possible. This same capability is also true for an external data item.

- Relaxed Time Stamp Checking (on page 48). It is no longer necessary for the compilation time stamp in the object program to match the **cobtoxml** time stamp in the template file. That is, the program may be recompiled without running the **cobtoxml** utility. It is necessary to run **cobtoxml** only when the relevant data structure(s) are changed.
- **UTF-8 Data Encoding**. Support has been added to both the UNIX and Windows implementations of XML Extensions to allow the in-memory representation of element content to use UTF-8 encoding. UTF-8 is a format for representing Unicode. This may be useful for COBOL applications that wish to pass UTF-8 encoded data to other processes. XML documents are normally encoded using Unicode. XML Extensions always exports XML documents with UTF-8 data encoding. For further information, see the applicable topics under:
  - Data Representation (on page 40), and
  - the discussion of XML and Character Encoding (on page 53)
- **New XML Statement**. A new XML statement has been added to XML Extensions that allows the developer to switch between the local character encoding (which is system-dependent) and the UTF-8 encoding format. See XML SET ENCODING (on page 85).

# Version 1

This was the initial release of the XML Toolkit. Version 1 of the XML Toolkit for RM/COBOL ran on Microsoft Windows 32-bit operating systems, excluding Windows 95.

The XML Toolkit for RM/COBOL is Liant Software Corporation's facility that allows RM/COBOL applications to access XML (Extensible Markup Language) documents. XML is the universal format for structured documents and data on the Web.

# Glossary of Terms

The glossary explains the terminology used throughout XML Extensions.

## Terminology and Definitions

The following terms are defined.

### Caching

Caching is a means of increasing performance by keeping loaded XSLT stylesheets, templates, and schema documents in memory for reuse without the need to reload them. If the application dynamically generates new copies of such documents, caching may be permanently or selectively disabled by the application. Caching is enabled by default at the beginning of an application.

### COBOL data structure

A COBOL data structure is a COBOL data item. In general, it is a group data item, but in some cases, it may be a single elementary data item. The **cobtoxml** utility, a component of XML Extensions, captures the COBOL data structure, including transformed data-names of the data items and subordinate data items, if any, so that a mapping between the COBOL data structure itself and an XML representation of the COBOL data structure can be accomplished in either direction at runtime.

### Document Type Definition (DTD)

The document type definition occurs between the XML header and the first element of an XML document. It optionally declares the document structure and entities. Declared entities may be referenced in the document.

## External XSLT stylesheet

An XSLT stylesheet that is provided by the user and referenced as a parameter in the XML EXPORT FILE/TEXT, XML IMPORT FILE/TEXT, or XML TRANSFORM statements. (The term "external" is used in this document to differentiate, where necessary, between the model file called the "internal XSLT stylesheet" and user-supplied "external" XSLT stylesheets.) See also XSLT stylesheet (on page 190).

## HTML

Hypertext Markup Language. A text description language related to SGML; it mixes text format markup with plain text content to describe formatted text. HTML is ubiquitous as the source language for Web pages on the Internet. Starting with HTML 4.0, the Unicode Standard functions as the reference character set for HTML content. See also SGML (on page 188), XHTML (on page 190), and XML (on page 190).

## Internal XSLT stylesheet

An XSLT stylesheet that is one of the model files created by the **cobtoxml** utility and applied automatically by the XML IMPORT FILE/TEXT statements when importing the content of a document into COBOL. See also XSLT stylesheet (on page 190).

## Model files

XML document files created by the **cobtoxml** utility. These include the example (*modelname*.xml), template (*modelname*.xtl), internal XSLT stylesheet (*modelname*.xls), and schema (*modelname*.xsd) files.

## Schema valid XML document

An XML document that conforms to a particular XML schema.

## SGML

Standardized Generalized Markup Language. A standard framework, defined in ISO 8879, for defining particular text markup languages. The SGML framework allows for mixing structural tags that describe format with the plain text content of documents, so that fancy text can be fully described in a plain text stream of data. See also HTML (on page 188) and XML (on page 190).

## Structured document

The term "structured document" describes the concept that a document can contain content, such as words, numbers, pictures, etc., as well as information describing the role of content elements and substructures. Adding "structure" to documents facilitates searching, sorting, or any one of a variety of operations to be performed on an electronic document. The benefits of adding structure to electronic documents include portability, re-usability, inter-system operability, ease-of-storage and

retrieval, longevity, quick access, and low distribution costs. XML is a set of rules for structuring a document using hierarchical markup.

## Stylesheet

See XSLT stylesheet.

## UNC

Universal Naming Convention.

## Unicode

Unicode was developed to support the worldwide interchange, processing, and display of diverse languages and technical disciplines of the world. Unicode is a character coding system that assigns a unique number to each character in each of the world's principal written languages. There exist several alternatives for how a sequence of such characters or their respective integer values can be represented as a sequence of bytes. The two most obvious encodings store Unicode text as either 2- or 4-byte sequences. The official terms for these encodings are UCS-2 and UCS-4, respectively. The current version of the Unicode Standard, developed by the Unicode Consortium, is v4.0.0. For an alternative encoding of Unicode, see also UTF-8 (on page 189).

## URL

Universal Resource Locator.

## UTF-8

UTF stands for Unicode Transformation Format. UTF-8 is an encoding scheme (that is, a method of mapping the Unicode code points to a digital representation), which is commonly used under Unix-style operating systems and in XML documents. Unicode is defined in ISO 10646-1:2000 Annex D and is also described in RFC 2279, as well as section 3.8 of the Unicode 3.0 standard. It is a variable length encoding scheme from 1 to 6 bytes per character. See also Unicode (on page 189).

## Valid XML document

See Schema valid XML document (on page 188).

## Well-formed XML document

A well-formed XML document is one that conforms to the syntax requirements of XML. A well-formed XML document may or may not be a valid document with respect to a particular XML schema.

## XHTML

Extensible HyperText Markup Language. When HTML 4.0 is expressed as XML, it is called XHTML. See also HTML (on page 188).

## XML

Extensible Markup Language. A subset of SGML constituting a particular text markup language for interchange of structured data. The Unicode Standard is the reference character set for XML content. See also Unicode (on page 189).

## XML schema

An XML schema is a document that specifies the structure and allowed content for another XML document.

## XSL

Extensible Stylesheet Language. A W3C standard defining XSLT stylesheets for (and in) XML. See also XSLT and W3C (on page 190).

## XSLT

Extensible Stylesheet Language for Transformations. XSLT is the "Transformations" part of the Extensible Stylesheet Language (XSL). A W3C standard, it is used to transform XML documents to other formats, including HTML, other forms of XML, and plain text. This powerful stylesheet language allows for more complex processing of the XML document's data. See also XSL and W3C (on page 190).

## XSLT stylesheet

An XML document that is written in the Extensible Stylesheet Language for Transformations. Note that XSLT stylesheets should not be confused with Cascading Style Sheets (CSS), which are a simple method for adding style, such as fonts, color, and spacing, to a document for final output to a browser; cascading style sheets are closely related to HTML and XHTML.

## W3C

World Wide Web Consortium. The main standards body for the World-Wide Web (WWW). W3C works with the global community to establish international standards for client and server protocols that enable online commence and communications on the Internet.

# Index

## E



## F



## G

## H



## I



## K



## L



## M



## N



## O



## P



## R

## S



## T



## U



## V

## W



## X