Liant Software Corporation

# XML Toolkit for RM/COBOL®

Version 1 for Windows®

LIANT

This manual is a user's guide for Liant Software Corporation's XML Toolkit, 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 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 |
| | (800) 762-6265 |
| Fax | (512) 343-9487 |
| Web site | http://www.liant.com/ |

Documentation History:
First Release XML Toolkit for RM/COBOL v1.0 (401217) December 2002 Rev 01/03

---

# Contents

# Preface

## Welcome to XML Toolkit for RM/COBOL

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.

The XML Toolkit has many capabilities. The major features support the ability to import and export XML documents to and from COBOL working storage. Specifically, the XML Toolkit 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 1.0 of the XML Toolkit for RM/COBOL runs on Microsoft Windows 32-bit operating systems, excluding Windows 95. A version is planned for UNIX platforms.

## About Your Documentation

The XML Toolkit for RM/COBOL documentation consists of a user's guide, which is distributed electronically in Portable Document Format (PDF) as part of the XML Toolkit software distribution CD-ROM.

**Note** To view and print PDF files, you need to install Adobe Acrobat Reader, a free program available from Adobe's Web site or from the software CD-ROM.

The *XML Toolkit for RM/COBOL 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 the XML Toolkit for RM/COBOL.

Chapter 2: Getting Started with XML Toolkit. This chapter presents the basic concepts used in the XML Toolkit 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 the XML Toolkit with RM/COBOL to develop an XML-enabled application.

Chapter 5: cobtoxml Utility Reference. This chapter describes the **cobtoxml** utility used by the XML Toolkit 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** dynamic link library used by the XML Toolkit for RM/COBOL.

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

Appendix B: XML Toolkit Sample Application Programs. This appendix provides information about the self-contained XML Toolkit sample application programs that are included with the development system in the XML Toolkit samples directory, **Samples**. Note that the most complete and up-to-date versions of the XML Toolkit sample programs can be found on the Liant Web site at http://www.liant.com/xmltk/samples.

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

The *XML Toolkit for RM/COBOL* manual also includes 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 CodeBridge 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 is preceded by a hyphen, then a leading hyphen must precede all options. When assigning a value to an option, the equal sign is optional if leading hyphens are used.

10. In the electronic PDF file, you may see this symbol displayed. It represents a "post-it" note that allows you to view last-minute comments about a specific topic on the page in which it occurs. This same information is also contained in the readme text file under the section, Documentation Changes.

    Double-click on the note symbols to open them. You can click the note window's Close box after you have reviewed the contents. These notes can be viewed in the Adobe Acrobat Reader but will not print, although you can copy and paste the text into another application, such as Microsoft Word, if you wish.

# 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 process, and provides a general overview of the XML Toolkit 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 the XML Toolkit for RM/COBOL. Depending on the complexity of your application, you may also need to know about XML style sheets.

## Installing XML Toolkit for RM/COBOL

Before you install the XML Toolkit for RM/COBOL (installation instructions begin on page 11), make sure that your computer configurations meets the following minimum hardware and software requirements, and that your XML Toolkit package contains the necessary items for development and deployment.

### System Requirements

The XML Toolkit hardware and software requirements are the same as RM/COBOL version 7.5 for 32-bit Windows, with the exception that Windows 95 is not supported by the XML Toolkit. Microsoft's XML parser MSXML 4.0 or greater also is required. (See the *RM/COBOL User's Guide for Windows*, version 7.5 or later.)

**Note** Windows 95 is not supported because the underlying XML parser (Microsoft's MSXML 4.0) is not supported on Windows 95.

The XML Toolkit may also be used in conjunction with Terminal Server.

It is highly recommended that you use Microsoft's Internet Explorer, version 6.0 or greater, as a convenient tool for viewing XML documents. (See the XML Toolkit README file for further details.)

For development, both the XML Toolkit development system and Liant's RM/COBOL 7.5 development system are required. For deployment, both the XML Toolkit deployment system and the RM/COBOL 7.5 runtime system are required.

## XML Toolkit for RM/COBOL Package

The XML Toolkit package contains the following items for development and deployment.

### *Development*

The XML Toolkit development system includes the following files:

- Deployment files. These files are listed in the next section.
- **cobtoxml** command line utility (**cobtoxml.exe**). See *Chapter 5: cobtoxml Utility Reference*, for more information.
- XML documents 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**).
- Example files. These programs or program fragments illustrate how **xmlif** library statements are used. (For further information, see *Appendix A: XML Toolkit Examples*. The example programs can be found in the XML Toolkit 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 Toolkit Sample Application Programs*, for more details. The sample application programs can be found in the XML Toolkit sample directory **Samples**.)

  **Note** The most complete and up-to-date versions of the XML Toolkit sample programs can be found on the Liant Web site at http://www.liant.com/xmltk/samples.

### *Deployment*

The XML Toolkit deployment system consists of the following files:

- **xmlif** COBOL-callable subprogram library (**xmlif.dll**). See *Chapter 6: xmlif Library Reference*, for more information.
- MSXML 4.0, the Microsoft XML parser (**msxml4.dll**, **msxml4a.dll**, and **msxml4r.dll**).

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

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" onpage 24.

## Installation

**Note** The XML Toolkit for RM/COBOL 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.

The following sections describe how to install the XML Toolkit for RM/COBOL development and deployment systems.

## *Installing the XML Toolkit for RM/COBOL Development System*

To install the XML Toolkit development system for Windows (**XMLTK10R.EXE**):

1. Restart Windows, and do not start any other applications.
2. Insert the XML Toolkit for RM/COBOL Installation CD into your CD 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 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 Toolkit 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 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 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 Liant XML Toolkit window.

## *Installing the XML Toolkit for RM/COBOL Deployment System*

The XML Toolkit for RM/COBOL deployment system, named **XMLTK10R.EXE**, is provided as a self-extracting executable that installs the deployment system components of the XML Toolkit.

The XML Toolkit deployment system is delivered on the XML Toolkit Development Installation CD as **redist\XMLTK10R.EXE** and is also available on the Liant Web site at http://www.liant.com/xmltk/redist/.

**Note** Your license for this product does not allow you to redistribute the entire XML Toolkit development system with your application. You may only redistribute the deployment system.

Provide the file, **XMLTK10R.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.

When the **XMLTK10R.EXE** file is executed, the Installation Directory dialog box is displayed. Follow these steps:

1. 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 Toolkit deployment system installation.

2. Click **Install** to continue.

3. When the installation completes, the Completion dialog box is displayed. Click **Close** to dismiss this dialog box.

# Introducing XML Toolkit for RM/COBOL

The XML Toolkit 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 Toolkit 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 the XML Toolkit (see *Chapter 5: cobtoxml Utility Reference*), 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.

By allowing standard COBOL data structures to be imported from and exported to XML documents, the XML Toolkit enables the direct processing and manipulation of XML-based electronic documents by the RM/COBOL application programmer. Furthermore, the XML Toolkit does this without requiring the application programmer to become thoroughly familiar with the numerous XML-related specifications and the extremely tedious 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 contents of a COBOL data structure may be used to generate an XML document with equal simplicity. The XML Toolkit 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 coded using the XML Style Sheet Language for Transformation (XSLT) can be applied at the same time. This powerful mechanism gives the XML Toolkit 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 an 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 application. Once the "document" content is described in this way, a simple command-line utility program (**cobtoxml.exe**), 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. At application execution time, a COBOL statement (accessed via a library of statements defined in copy files supplied with the XML Toolkit) is used to call a subprogram (**xmlif.dll**), referenced throughout this document as the **xmlif** library, which implements the complete runtime functionality of the XML Toolkit. For more information, see *Chapter 5: cobtoxml Utility Reference*, and *Chapter 6: xmlif Library Reference*.

## 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 has specified a markup-based language called XML. As a close cousin of HTML, it 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. 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 (which is probably not why the COBOL programmer cares about it), 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.

Most of all, XML should be viewed as 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 must know more about the data. Specifically, you would have to know how it is structured, and what the sizes of the fields are. 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 as 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 beginning “tag” (the “<contact>”) and an ending tag (the “</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 firstname pic x(10).
   10 lastname pic x(10).
   10 address.
      20 streetaddresslines pic 9.
      20 streetaddresses.
         30 streetaddress occurs 1 to 9 times
               depending on streetaddresslines pic x(20).
      20 city pic x(20).
      20 state pic x(2).
      20 postalcode pic 9(5).
   10 email 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 more like 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>
  <firstname>Betty</firstname>
  <lastname>Smith</lastname>
  <address>
         <streetaddresslines>3</streetaddresslines>
         <streetaddresses>
                <streetaddress>Knox College</streetaddress>
                  <streetaddress>Campus Box 9999</streetaddress>
                <streetaddress>2 E. South St.</streetaddress>
                </streetaddresses>
         <city>Galesburg</city>
         <state>IL</state>
         <postalcode>61401</postalcode>
  <email>bs@aol.com</email>
</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>
```

There 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 called “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 record contents. 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 help of an important XML tool called a “style sheet.” For now, assume that a style sheet 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.

The example document, after style sheet transformation, might look like this:

```
<contact>
   <attr-type>student</attr-type>
      <firstname>Betty</firstname>
      <lastname>Smith</lastname>
      <address>
               <attr-form>US</attr-form>
             <streetaddresslines>3</streetaddresslines>
             <streetaddresses>
                    <streetaddress>Knox College</streetaddress>
                    <streetaddress>Campus Box 9999</streetaddress>
                    <streetaddress>2 E. South St.</streetaddress>
               </streetaddresses>
             <city>Galesburg</city>
             <state>IL</state>
             <postalcodegroup zipplus4="N">
                    <attr-zipplus4>N</attr-zipplus4>
                    <postalcode>61401</postalcode>
               </postalcodegroup>
   <email>bs@aol.com</email>
</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 style sheet, and you now have a document with a direct equivalent in COBOL:

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

# Chapter 2: Getting Started with XML Toolkit

This chapter presents the basic concepts used in the XML Toolkit for RM/COBOL by creating an example XML-enabled application.

## 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 contents of a COBOL data record and you may view the text of an XML document. In XML, markup is used both to name and to describe the text elements of a document. In COBOL, the data structure itself provides names and descriptions of the elements within a document.

The XML Toolkit 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 except for the Linkage Section or externals. Specifically, the XML Toolkit 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.

The XML Toolkit consists of the following two main components:

- **cobtoxml** utility
- **xmlif** library

The **cobtoxml** utility, **cobtoxml.exe**, runs as a post-compile step. This program creates a set of XML documents, called model files (see page 24), which describe a selected COBOL data structure as a set of XML documents. The **xmlif** library, **xmlif.dll**, 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 Toolkit for RM/COBOL development system and the RM/COBOL development system (version 7.5 or later) are installed on your computer.

**Note** More examples and information about complete sample application programs can be found in *Appendix A: XML Toolkit Examples*, *Appendix B: XML Toolkit Sample Application Programs*, and in the XML Toolkit examples and samples directories (**Examples** and **Samples**, respectively). The most up-to-date versions of the XML Toolkit sample programs can be found on the Liant Web site at http://www.liant.com/xmltk/samples.

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

1. Design the Data Structure. Develop a COBOL program, or modify an existing one, using statements that refer to the **xmlif** library.
2. Compile the Program. 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**.** 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. Test the program and repeat steps 1 through 4 as necessary.
5. Deploy the Application. After stripping the symbol table information from the COBOL object program, distribute the XML Toolkit 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 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 structure contains only one numeric element: the zip code. For demonstration purposes, it is represented as binary.

## Compile the Program

In step 2, 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 the XML Toolkit.

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. In this case, the following files are created:

- Example file (**getstarted.xml**)
- Template file (**getstarted.xtl**)
- Schema file (**getstarted.xsd**)
- Style sheet (**getstarted.xsl**)

### Example File

The XML document, **getstarted.xml**, is an example file created primarily for the COBOL developer's reference. 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's 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 our example, `<root>` marks a beginning and `</root>` marks an ending. The tags `<customer-address>` and `<address-3>` have both start-tags and end-tags as well. XML also 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 the document, you will note that the shorthand sequence is not used by this example.

This document contains no text, only markup, as it is intended only as a reference for the programmer. 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. Since 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 1.0d.00 for RM/COBOL version 7.50 or greater on:
  Wed Nov 20 12:34:19 2002 -->
<!-- data item "customer-address" in program "GETTING-STARTED" in file
  "C:\xmlexample\getstarted.cob" -->
<root type="nonnumeric" kind="GRP" compiledTimeStamp="2002-11-20T12:34:12"
 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>
```

## Style Sheet File

The XML document, **getstarted.xsl**, is an internal style sheet. Style sheets 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 style sheet 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 style sheet 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 style sheets.

### Schema File

The XML document, **getstarted.xsd**, is a schema file used to validate the contents 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. The **cobtoxml** utility has an option to disable the generation of a schema file (see "Schema Options" on page 59 in *Chapter 5: cobtoxml Utility Reference*), and the **xmlif** library has options to validate or not to validate the contents of an XML document (see the descriptions of XML VALIDATE FILE on page 70 and XML VALIDATE TEXT on page 71 in *Chapter 6: xmlif Library Reference*).

The schema file 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 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 step 4, you execute and test the program.

In the following sections, you are going to build upon the preceding steps by adding more functionality to the COBOL data structure designed in step 1 of this example. Then, steps 2 and 3 are repeated as necessary.

First, 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 contents of the 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 a fragment of the program. It was just enough to show the data structure and allow program compilation so that it would be possible to examine the model files generated by the **cobtoxml** utility.

The **xmlif** library is a COBOL-callable subprogram. The interface to the library 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 the XML Toolkit.

In order to call the **xmlif** library, the COBOL program fragment from step 1 is further developed by adding the following lines (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) 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 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, it is necessary that you run the **cobtoxml** utility again, even if the data structure has not changed. This is because the **xmlif** library must have access to 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 contents 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 contents of a 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`). 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
runcobol getstarted
```

This time the program creates an XML document in the 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 are 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

Use the RM/COBOL Combine Program Utility, **rmpgmcom**, which comes 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
```

The model file documents contain a time stamp that reflects the compilation date and time of the COBOL object file. If you recompile the COBOL source to remove the symbol table, the time stamp of the model files will not match the compilation date and time and the **xmlif** library will generate an error message.

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 the XML Toolkit, install the XML Toolkit deployment system on each platform that runs the application. You may do this using the XML Toolkit Installation disk.

# 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:

- File Management
- Data Conventions
- Copy Files
- Limitations
- Optimizations

## File Management

The management of data files when using the XML Toolkit is similar, but not identical, to other RM/COBOL data file management issues. The sections that follow discuss these differences.

### Automatic Search for Files

During development with the XML Toolkit, 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 pathname is not provided by the XML Toolkit when the **xmlif** library locates files. That is, the XML Toolkit 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 model files and style sheet files, as necessary.

## File Management Conventions

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

### *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 (**.xml**, **.xsl**, .**xtl**, and sometimes **.xsd**). For more information, see "Model Files" on page 24 in *Chapter 2: Getting Started with XML Toolkit*, and "Referencing XML Model Files" on page 59 in *Chapter 5: cobtoxml Utility Reference*.

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.

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:").

## External Style Sheet File Naming Conventions

In addition to the style sheet that is produced as part of the model files, other style sheets 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 the style sheet file (with the **.xsl** extension added) *except* when:

- the style sheet 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:").

## 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 the XML Toolkit, several suppositions were made about data transformations between COBOL and XML, including those relating to the following items:

- Data Representation
- FILLER Data
- Missing Intermediate Parent Names
- Sparse COBOL Records

## 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 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.

## FILLER Data

Unnamed data description entries, referred to as FILLER data 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 will not reliably produce well-formed XML sequences. For this reason, FILLER data 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, however, is 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, 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 are present in a data item that is referenced by the XML EXPORT statement, the resulting document is validated 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 the XML Toolkit to reconstruct missing intermediate parent names in a COBOL data structure. There are two ways in which these missing names may be generated:

- One technique is to determine whether the element name is unique. If this is true, then the intermediate parent names are generated by the model file style sheet.
- The other method is 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.

The following sections illustrate the two approaches.

### *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, you use the **-sn** (schema none) option on the **cobtoxml** utility.. You may also delete an existing schema model file (**.xsd** extension).

### *Unique Identifier (uid)*

The unique identifier (uid) attribute is generated by an XML EXPORT FILE or XML EXPORT TEXT statement if XML attributes are enabled. Attributes may be enabled by using the XML ENABLE ATTRIBUTES (page 83) statement before the XML EXPORT statement.

Using the same COBOL data structure illustrated on page 41 for unique element names, 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="2002-12-04T10:57:22"
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 here. 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 a style sheet or other process and then added again before the XML IMPORT 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, you use the **-sn** (schema none) option on the **cobtoxml** utility. You may also delete an existing schema model file (**.xsd** extension).

## 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** extension).

# Copy Files

Under most circumstances, you should make use of the copy files that are provided in the XML Toolkit. This section describes the various points to consider when using copy files with the XML Toolkit, including:

- Statement Definitions
- Displaying Status Information
- Application Termination

## 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. Do not modify the contents of this copy file or the copy files that it copies (**lixmdef.cpy** and **lixmlrpl.cpy**).

## 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.

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 the XML Toolkit and the second PERFORM statement displays any status from the XML TERMINATE operation.

The following logic is sufficient to successfully terminate the XML Toolkit:

```
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.

# Limitations

This section describes the limitations of the XML Toolkit 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)
- Edited Data Items
- Wide and Narrow Characters
- Data Item Size
- OCCURS Restrictions
- Reading, Writing, and the Internet

## Data Items (Data Structures)

Data items that are passed to the XML Toolkit 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 XML IMPORT or XML EXPORT operations (see pages 62 through 66).

The XML IMPORT and XML 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 statement and retrieve data from the data item after using the XML IMPORT 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 items must be nonnumeric.

## 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 the XML Toolkit is represented in UTF-8 format, which is essentially ASCII with extensions for representing 16-bit 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 directly write or export an XML document 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.

## Occurs Depending

As expected, on output, the XML EXPORT statement 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 (see "Empty Occurrences").

On input, the XML IMPORT statement will store the value of the DEPENDING variable. The XML IMPORT statement 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 is generated by the **cobtoxml** utility, then the schema will report an error if not all of the elements specified by the DEPENDING variable are present.

## Empty Occurrences

On output, the XML EXPORT statement recognizes 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 (XML IMPORT), a schema may detect an error if not all expected occurrences of an item are present. In order to prevent this, you may enable all occurrences (use the XML DISABLE ALL-OCCURRENCES statement, described on page 81) when generating the document with XML EXPORT.

## Cached XML Documents

Since XML style sheets and template files are largely invariant, performance can usually be improved by caching previously loaded style sheets and templates in memory.

For some applications, it may be useful to disable caching. If style sheets and template files are generated or replaced in real time, then the cached files 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.

The XML ENABLE CACHE, XML DISABLE CACHE, and XML FLUSH CACHE statements may be used to enable or disable document caching. By default, caching is enabled. For more information on these XML statements, see pages 83 and 84 in *Chapter 6: xmlif Library Reference*.

# Chapter 4: XML Considerations

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

- Character Encoding
- Style Sheets
- Schemas

## Character Encoding

XML documents use the Unicode character encoding standard internally. 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 the XML Toolkit 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.

# Style Sheets

Style sheets are used to transform an XML document into another document (not necessarily XML format, such as HTML, PDF, RTF, and so forth). A style sheet is itself an XML document. The **xmlif** library has a specific statement for performing style sheet transformations (XML TRANSFORM FILE). In addition, the XML IMPORT and XML EXPORT statements allow a style sheet to be specified as a parameter, making it possible to transform a document while importing or exporting XML documents. For more information on these XML statements, see *Chapter 6: xmlif Library Reference*.

The format of XML documents generated by the XML Toolkit 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 XML style sheet to transform between the COBOL-generated XML document format and the expected document format. XML style sheets are extremely powerful. You may wish to use a style sheet editing tool to design your style sheets (for example, Microsoft's BizTalk Mapper, which is part of BizTalk Server 2000).

Keep in mind that style sheets are unidirectional. Therefore, it is possible that you will have to design two style sheets 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 format.

# Schemas

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 in any of the following three ways:

- 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); or
- A process that loads a given XML document may also load a schema file that controls the document.

The third approach is used by the **xmlif** library. 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. This validation can be skipped by not having the **cobtoxml** utility generate a schema file by specifying the **-sn** (schema none) option or by simply deleting the schema file.

# Chapter 5: cobtoxml Utility Reference

This chapter describes the **cobtoxml** utility used by the XML Toolkit 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 a 32-bit console application. It processes the symbol table of a previously compiled RM/COBOL object file and produces a set of XML documents. These documents are often called XML model files and are described in the "Referencing XML Model Files" topic that begins on page 59. (See also "Model Files" on page 24 in *Chapter 2: Getting Started with XML Toolkit*.)

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.

# Command Line Interface

The **cobtoxml** utility is executed with the following command:

```
cobtoxml cob-file-name data-item-name [xml-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 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.

*xml-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**, **.xsl**, .**xtl**, 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 beginning on page 57. 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** When no command line parameters are entered, the following **cobtoxml** usage message is displayed (including the error) as follows:

```
Error: 33[0] - parameter - COBOL object file name missing

RM/COBOL cobtoxml utility - Version 1.00.00 for 32-Bit Windows.
Copyright (c) 2001-2002 by Liant Software Corp.  All rights reserved.

Usage:  cobtoxml cob-file-name data-item-name xml-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
  xml-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 Options*

Banner options control the amount of information displayed during the execution of the **cobtoxml** utility and are created by entering a hyphen character followed by the letter “b” and then either the letter “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*

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.

Name options are generated by entering a hyphen character (-) followed by the letter “n” and then one or more of the following letters (in any order): “a”, “f”, “h”, “l”, “m”, “p”, and “u”.

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

| Option | Description |
|---|---|
| a (After parent) | This option is used to ensure 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. This is done until the tag name becomes unique. |
| f (First) | The first letter of the tag name is capitalized. |
| h (Hyphen out) | Hyphen characters in the COBOL data-name are removed from the tag name. |
| l (Lowercase) | Unless overridden by the options "f" or "m", all characters in the tag name are lowercase. |
| m (Mixed case) | The first letter after a hyphen character in the COBOL data-name is represented as uppercase in the generated tag name. |
| p (Prefix out) | All characters in the COBOL data-name up to and including the first hyphen are removed. This option is useful where all data items in a structure begin with the same sequence. However, this option should be used with care. If the item name contains no hyphen characters then the generated tag name will be empty. |
| u (Uppercase) | All characters in the COBOL data-name are represented as uppercase in the generated tag name. |

### *Schema Options*

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.

Schema options are generated by entering a hyphen character followed 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 may be referenced by the COBOL application via a traditional path name or by an Internet address. More information about model files can be found in the section, “Model Files” in *Chapter 2: Getting Started with XML Toolkit*. Examples of references to XML model files are shown in the following table:

| Filename | Type of Referencing |
|---|---|
| c:\myfiles\myapp.xml | Simple pathname. |
| \\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 for each data structure that is specified. These XML documents are the internal style sheet, the template file, the example file, and the schema file. The example file is generated as a reference for the developer. The internal style sheet, the template file, and the optional schema file are used internally by the COBOL application.

## Internal Style sheet

The internal style sheet (a file having the **.xsl** extension) is an XML style sheet. It adds COBOL-like attributes to an existing XML document. The **xmlif** library uses the internal style sheet when importing an XML document.

## 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.

## 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 style sheets that may be needed.

## 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.

# Chapter 6: xmlif Library Reference

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

## What is the xmlif Library?

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

The **xmlif** library uses the Microsoft MSXML 4.0 parser.

The following sections describe the various types of statements used by the **xmlif** library:

- Document Processing Statements (see page 62). These statements are used to process, manipulate, or validate XML documents.
- Document Management Statements (see 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 (see page 75). These statements are useful when implementing directory-polling schemes.
- State Management Statements (see 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

Several types of statements are used to process, manipulate, or validate XML documents:

- Export statements (XML EXPORT FILE and XML EXPORT TEXT) 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.
- Import statements (XML IMPORT FILE and XML IMPORT TEXT) are available to convert the content of an XML document—either an external file or an internal text string—to a COBOL data item.
- Test and validation statements (XML TEST WELLFORMED-FILE, XML TEST WELLFORMED-TEXT, XML VALIDATE FILE, and XML VALIDATE TEXT) are available to verify that an XML document—either an external file or an internal text string—is well formed or valid.
- In addition, the XML TRANSFORM FILE statement is provided to create an XML document (as an external file) using a style sheet (also an external file). This statement also may be used to generate files that are not XML documents, such as HTML, PDF, RTF files, and so forth.

## XML EXPORT FILE

This statement has the following parameters:

| Parameter | Description |
|---|---|
| *DataItem* | The name of a 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 24 in *Chapter 2: Getting Started with XML Toolkit*. |
| [*StyleSheetName*] | Optional. The name of a style sheet that will be used to transform the generated XML document before it is stored. |

## 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 and then output to the file specified by the *DocumentName* parameter. If the optional *StyleSheetName* parameter is present, the style sheet 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 a Style Sheet:*

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

*With a Style Sheet:*

```
XML EXPORT FILE
    MY-DATA-ITEM
    "MY-DOCUMENT.XML"
    "MY-MODEL-FILE"
    "MY-STYLE-SHEET"
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 24 in *Chapter 2: Getting Started with XML Toolkit.* |
| [*StyleSheetName*] | Optional. The name of a style sheet that will be used to transform the generated XML document before it is stored. |

## 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 output as a text string. The address of the text string is placed in the COBOL pointer data item parameter specified by *DocumentPointer*. If the optional *StyleSheetName* parameter is present, the style sheet 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 the XML FREE TEXT statement (see page 72).

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

### *Examples*

*Without a Style Sheet:*

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

*With a Style Sheet:*

```
XML EXPORT TEXT
    MY-DATA-ITEM
    MY-DOCUMENT-POINTER
    "MY-MODEL-FILE"
    "MY-STYLE-SHEET"
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 24 in *Chapter 2: Getting Started with XML Toolkit*. |
| [*StyleSheetName*] | Optional. The name of a style sheet 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* is present, the style sheet 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 and 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 a Style Sheet:*

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

*With a Style Sheet:*

```
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 24 in *Chapter 2: Getting Started with XML Toolkit*. |
| [*StyleSheetName*] | Optional. The name of a style sheet 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* is present, the style sheet 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 and 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 a Style Sheet:*

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

*With a Style Sheet:*

```
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 parameters:

| 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.

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 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. |

## 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.

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 style sheet 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 style sheet 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 style sheet.

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.

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

### *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.

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

### *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 statements include XML FREE TEXT, XML GET TEXT, XML PUT TEXT, and XML REMOVE FILE.

## 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 (see page 72) 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 and XML GET UNIQUEID.

Directory polling (as related to XML documents) is a technique that two or more independent processes can use to pass XML documents between processes. 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 be given a unique name. A reader process finds, processes and removes XML documents from the same well-known directory.

Directory polling is a technique that may be used to communicate with Microsoft's BizTalk server and other message-driven communications systems. It is a technique that also may 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. 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** Problems have been encountered on Windows systems running the older FAT32 file system. These systems include Windows 98 and Windows Me.

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 the XML Toolkit.

When a large number of XML document files are written to a directory by the XML Toolkit (using the XML EXPORT FILE statement described on page 62), it is possible that files will not be placed in the directory and no error will be returned by the operating system either to the XML Toolkit 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 the XML IMPORT FILE statement (see page 65).

Before calling the XML FIND FILE statement again (to process the next file), the statement XML REMOVE FILE (see page 74) should be called 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 of a 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. |

## 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

Several states or conditions of the **xmlif** library are controlled by calls to the following XML statements:

- Initialization and termination. Before issuing a call to any other **xmlif** library statement, the XML INITIALIZE statement (see page 80) must be called. Similarly, the XML TERMINATE statement (see page 80) must be called when the COBOL application is finished using the **xmlif** library.

- Empty array occurrences. As an optimization, trailing "empty" occurrences of arrays are normally not generated by the statements, XML EXPORT FILE or XML EXPORT TEXT (see pages 62 and 64, respectively).

  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 the XML DISABLE ALL-OCCURRENCES statement (see page 81). It is possible to force all occurrences to be output by calling the XML ENABLE ALL-OCCURRENCES statement (see page 82).

- COBOL attributes. For each element generated by the statements, XML EXPORT FILE or XML EXPORT TEXT (see pages 62 and 64, respectively), 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 a style sheet transformation) to have access to these attributes (specifically, length and subscript are often interesting to a follow-on activity). Using the XML DISABLE ATTRIBUTES statement (see page 82) does not allow attributes to be written (this is the default). Using the XML ENABLE ATTRIBUTES statement (see page 83) forces these attributes to be written.

- Document caching. Some XML documents such as style sheets and the model files (the XML template file, internal style sheet, and schema files) are normally considered to be static. That is, they are generated when the application is developed and are not modified until the application is modified.

  As a performance optimization, when the **xmlif** library loads a style sheet or model file it is cached (retained in memory) for an indefinite period of time. This is the default behavior. Files in the cache may be flushed from memory if the cache is full and an additional style sheet or model file is required for the current operation.

  If style sheets are being generated dynamically, caching may be selectively enabled or disabled. Executing the XML ENABLE CACHE statement (see page 84), which is the default behavior, enables caching of style sheets and model files. Executing the XML DISABLE CACHE statement (see page 83) forces style sheets and model files to be loaded each time they are referenced. Executing the XML FLUSH CACHE statement (see page 84) flushes all style sheets and model files from memory 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, XML EXPORT TEXT, XML IMPORT FILE, and XML IMPORT TEXT (see pages 62 through 66), 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.

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

# 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 is the required version (7.5 or greater) and retrieves required information from the runtime system. RM/COBOL runtime version 7.5 or greater is required because information needed by the **xmlif** library is not available in prior runtime versions. The underlying XML parser is initialized.

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 7.5 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 closes a session with the **xmlif** library. The interface to the underlying XML parser is 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 and XML EXPORT TEXT (see pages 62 and 64, respectively). 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 and XML ENABLE ATTRIBUTES (see pages 82 and 83, respectively). If attributes are enabled (XML ENABLE ATTRIBUTES has been called), then all empty occurrences are not generated. If attributes are disabled (the default state or if the XML DISABLE ATTRIBUTES statement 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 and XML EXPORT TEXT (see pages 62 and 64, respectively), 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 style sheet.

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 XML style sheets and model files.

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 XML style sheets and model files.

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 XML style sheets and model files.

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 the XML FIND FILE statement 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.
```

# 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.

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 Toolkit Examples

This appendix contains a collection of programs or program fragments that illustrate how **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 the XML Toolkit for RM/COBOL. More examples can be found in the XML Toolkit examples directory (**Examples**).

**Note** You will find it instructive to examine these examples first before referring to *Appendix B: XML Toolkit Sample Application Programs*, which describes how to use and access the more complete sample application programs that are included with the XML Toolkit 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 page 190).

# Example 1: Export File and Import File

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

This example uses the following XML statements (for more information about the **xmlif** library, see *Chapter 6: xmlif Library Reference*):

- XML INITIALIZE (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (page 62). The XML EXPORT FILE statement constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE(page 65). The XML IMPORT FILE statement reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (page 80). The XML TERMINATE statement 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 on model files, see "Model Files" in *Chapter 2: Getting Started with XML Toolkit*. For more information on the **cobtoxml** utility, see *Chapter 5: cobtoxml Utility Reference*.

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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example1.cob**. The K Option "kills" 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 in line 4 instructs Windows to start the runtime and then wait (the /w Option) for its completion. 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 contents 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. It is possible for XML INITIALIZE to fail; therefore, 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 contents of the XML document are 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 contents of the data item defined in the copy file, **liant.cpy**, are 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 also 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 contents of this data item are 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 Toolkit 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-name Z, so that any error condition is obtained here via a GO TO 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

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 example1**) produces the following display. Note that pressing a key will terminate the program.

```
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 use IE to 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 contents 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 Style Sheets

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

This example is almost identical to "Example 1: Export File and Import File" (see page 88). However, an XSLT style sheet is used to transform the exported document into a different format. Similarly, when the document is imported, a different style sheet is used to reformat the document into the form that is expected by COBOL. (See page 102 for more information on style sheets.)

This example uses the following XML statements:

- XML INITIALIZE (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (page 62). The XML EXPORT FILE statement constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE (page 65). The XML IMPORT FILE statement reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (page 80). The XML TERMINATE statement terminates or closes the session with the **xmlif** library.

**Note** The XML EXPORT FILE and XML IMPORT FILE statements each contain an additional parameter: the name of the style sheet being used for the transform.

## 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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example2.cob**. The K Option "kills" 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 contents 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. It is possible for XML INITIALIZE to fail; therefore, 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 contents of the XML document are 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 contents of the data item defined in the copy file, **liant.cpy,** are 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 also 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 contents of this data item are 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 Toolkit 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 style sheet 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 style sheet 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-name Z, so that any error condition is obtained here via a GO TO 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. |

## Style Sheets

The two style sheets used in this example are for reference only (a tutorial on style sheet development is outside the scope of this document). 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 the COBOL internal format.

These style sheets are manually generated using a text editor program. Other tools, such as Microsoft's BizTalk Mapper, may be used to generate style sheets.

### 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

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 example2**) produces the following display. Note that pressing a key will terminate the program.

```
Example-2 - Illustrate EXPORT FILE and IMPORT FILE with style sheets
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 use IE to 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 contents 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". 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 contents of a COBOL data item. Then the program reads (or imports) the same XML document and places the contents in the same COBOL data item.

This program is very similar to "Example 1: Export File and Import File" (see 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 (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (page 62). The XML EXPORT FILE statement constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE (page 65) The XML IMPORT FILE statement reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (page 80). The XML TERMINATE statement 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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example3.cob**. The K Option "kills" 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 contents 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. It is possible for XML INITIALIZE to fail; therefore, 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 contents of the XML document are 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 contents of the data item defined in the copy file, **liant3.cpy**, are 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" (see page 88) 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 also contain the current time.

## Other Definitions

Include the copy file, **lixmlall.cpy**, 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 contents of this data item are 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 Toolkit 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-name Z, so that any error condition is obtained here via a GO TO 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

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 example3**) produces the following display. Note that pressing a key will terminate the program.

```
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 use IE to 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 contents 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.

The program first writes (or exports) several XML document files from the contents 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 contents in the same COBOL data item.

This example uses the following XML statements:

- XML INITIALIZE (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (page 62). The XML EXPORT FILE statement constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE (page 65). The XML IMPORT FILE statement reads an XML document (from a file) into a COBOL data item.
- XML ENABLE ATTRIBUTES (page 83). The XML ENABLE ATTRIBUTES statement 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 below), 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 (page 82). The XML DISABLE ATTRIBUTES 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 (page 82). The XML ENABLE ALL-OCCURRENCES statement causes all occurrences of a data item to be exported to an XML document.

- XML DISABLE ALL-OCCURRENCES (page 81). The XML DISABLE ALL-OCCURRENCES statement 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 (page 80). The XML TERMINATE statement 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 Utility, **rmpgmcom**.

## 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.

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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example4.cob**. The K Option "kills" 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 contents 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. It is possible for XML INITIALIZE to fail; therefore, the return status must be checked before continuing.

Data is exported from the data item `Data-Table` 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 contents of these four XML documents (plus two additional "pre-created" XML documents, **table5.xml** and **table6.xml**) are 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 contents are described beginning on page 121.

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 contents of the data item defined in the copy file, **liant.cpy,** are 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 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 contents of this data item are 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 Toolkit 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-name Z, so that any error condition is obtained here via a GO TO 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

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 example4**) produces the following display. Note that pressing a key will terminate the program.

```
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 use IE to 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** are supplied with the example, and they also describe the same COBOL content.

The only non-empty occurrences are for the second and fourth elements of the array. The contents 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 contents 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. 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 contents 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 contents 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 contents 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, a style sheet 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

The program first writes (or exports) an XML document as a text string from the contents of a COBOL data item. Then the program reads (or imports) the same XML document and places the contents 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 (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT TEXT (page 64). The XML EXPORT TEXT statement constructs an XML document (as a text string) from the contents of a COBOL data item.
- XML IMPORT TEXT (page 66). The XML IMPORT TEXT statement reads an XML document (from a text string) into a COBOL data item.
- XML PUT TEXT (page 73). The XML PUT TEXT statement copies an XML document from a text string to a data file.
- XML FREE TEXT (page 72). The XML FREE TEXT statement releases the memory that was allocated by XML EXPORT TEXT to hold the XML document as a text string.
- XML TERMINATE (page 80). The XML TERMINATE statement 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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example5.cob**. The K Option "kills" 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 contents 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" (see 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. It is possible for XML INITIALIZE to fail; therefore, 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 contents of the XML document are 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 sole reason for using the XML PUT TEXT statement is to make the contents 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 contents of the data item defined in the copy file, **liant.cpy,** are 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 also 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 contents of this data item are 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 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 Toolkit 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`<br>`  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-name Z, so that any error condition is obtained here via a GO TO 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

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 example5**) produces the following display. Note that pressing a key will terminate the program.

```
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 use IE to 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 contents 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.

The program first writes (or exports) five XML document files from the contents 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 contents in the COBOL data item.

This example uses the following XML statements:

- XML INITIALIZE (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (page 62). The XML EXPORT FILE statement constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE (page 65). The XML IMPORT FILE statement reads an XML document (from a file) into a COBOL data item.

- XML TERMINATE (page 80). The XML TERMINATE statement terminates or closes the session with the **xmlif** library.
- XML GET UNIQUEID (page 77). The XML GET-UNIQUEID statement is used to generate a unique identifier that can be used to form a filename.
- XML FIND FILE (page 76). The XML FIND FILE statement finds a XML document file in the specified directory (if one is available).
- XML REMOVE FILE (page 74). The XML REMOVE FILE statement 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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example6.cob**. The K Option "kills" 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. It is possible for XML INITIALIZE to fail; therefore, the return status must be checked before continuing.

The current time, which will become the contents 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 pathname 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 contents of the 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 contents of this data item are 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 Toolkit 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` | Generate a unique identifier. |
| `If Not XML-OK Go to Z.` | If the statement terminates unsuccessfully, go to the termination logic. |
| `Move Spaces to Unique-File-Name`<br>`String "Stamp\A" delimited by size`<br><br>`  Unique-Name delimited by SPACE`<br><br>`  ".xml" delimited by size`<br>`  into Unique-File-Name.` | Convert the unique identifier into a pathname. |
| `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`<br>`    XML FIND FILE`<br>`      "Stamp"`<br>`       Unique-File-Name`<br>`    If XML-IsSuccess`<br>`      Exit Perform`<br>`    End-If`<br>`    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`<br>`    If Not XML-OK`<br>`      Go to Z`<br>`    End-If`<br>`  End-Perform` | Inner perform loop. Iterate until Exit Perform<br>Execute XML FIND FILE parameters:<br>directory name<br>and filename<br>If the statement returned success,<br>exit the paragraph<br>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.<br>If the statement terminates unsuccessfully,<br>go to the termination logic.<br><br>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"`<br>`  If Not XML-OK Go to Z End-If` | Import the file that was found using:<br>the data item,<br>the filename,<br>and the model filename.<br>If the statement terminates unsuccessfully, go to the termination logic. |
| `  XML REMOVE FILE`<br>`      Unique-File-Name`<br>`  If Not XML-OK Go to Z End-If` | Remove the file that has just been processed;<br>otherwise, find it again.<br>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-name Z, so that any error condition is obtained here via a GO TO 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

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 is after exporting five documents to the **Stamp** directory. The second display is after polling the **Stamp** directory and importing the five documents.

### *First Display*

Note that 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 use IE to display the 'Stamp' directory

Press a key to continue:
```

### *Second Display*

Note that pressing a key will terminate the program.

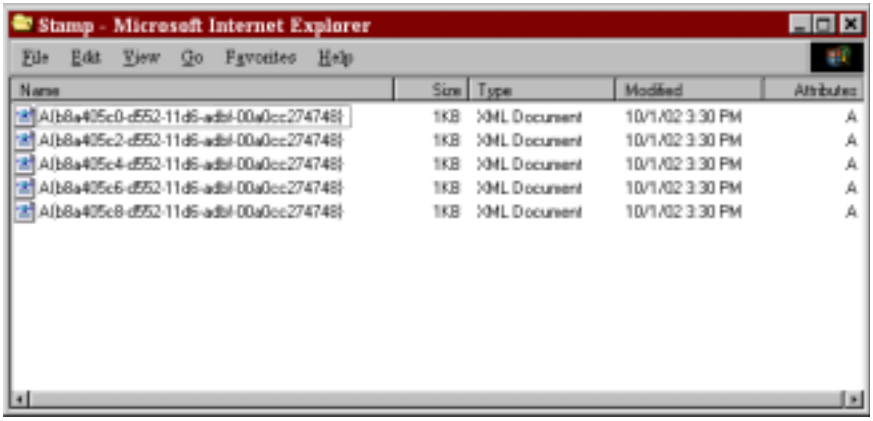
```
E:\xmlexample\Stamp\A{b8a405c0-d552-11d6-adbf-00a0cc274748}.xml imported
by XMLImport
Contents: 15303258
 E:\xmlexample\Stamp\A{b8a405c2-d552-11d6-adbf-00a0cc274748}.xml imported
by XMLImport
Contents: 15303264
 E:\xmlexample\Stamp\A{b8a405c4-d552-11d6-adbf-00a0cc274748}.xml imported
by XMLImport
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
2002/10/\
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 contents.




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 contents of an XML document may be verified.

The program first writes (or exports) an XML document file from the contents of a COBOL data item. Then the program verifies that the generated document is well formed. Finally, the program verifies that the contents of the document conform to the schema file that was generated by the **cobtoxml** utility.

This example uses the following XML statements:

- XML INITIALIZE (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (page 62). The XML EXPORT FILE statement constructs an XML document (as a file) from the contents of a COBOL data item.
- XML TEST WELLFORMED-FILE (page 67). The XML TEST WELLFORMED-FILE statement verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE FILE (page 70). The XML VALIDATE FILE statement verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML TERMINATE (page 80). The XML TERMINATE statement 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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example7.cob**. The K Option "kills" 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. It is possible for XML INITIALIZE to fail; therefore, 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 contents of **liant7.xml** are 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 contents of the data item defined in the copy file, **liant.cpy,** are 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 also 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 contents of this data item are 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 Toolkit 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 "Liant7" "Example7".` | Execute the XML VALIDATE FILE statement specifying: the XML document filename 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-name Z, so that any error condition is obtained here via a GO TO 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

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 example7**) produces the following display. Note that pressing a key will terminate the program.

```
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 use IE to 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 contents 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 contents of an XML document may be verified. Next, the program verifies that the generated document is well formed. Finally, the program verifies that the contents of the document conform to the schema file that was generated by the **cobtoxml** utility.

This example uses the following XML statements:

- XML INITIALIZE (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT TEXT (page 64). The XML EXPORT TEXT statement constructs an XML document (as a text string) from the contents of a COBOL data item.
- XML TEST WELLFORMED-TEXT (page 68). The XML TEST WELLFORMED-TEXT statement verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE TEXT (page 71). The XML VALIDATE TEXT statement verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML PUT TEXT (page 73). The XML PUT TEXT statement copies an XML document from a text string to a data file.
- XML FREE TEXT (page 72). The XML FREE TEXT statement releases the memory that was allocated by XML EXPORT TEXT to hold the XML document as a text string.
- XML TERMINATE (page 80). The XML TERMINATE statement 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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example8.cob**. The K Option "kills" 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. It is possible for XML INITIALIZE to fail; therefore, 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 contents of the generated XML document are 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 sole reason for using the XML PUT TEXT statement is to make the contents 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 contents of the data item defined in the copy file, **liant.cpy,** are 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 also 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 contents of this data item are 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 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 Toolkit 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: the XML document text name 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>`"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`<br>`"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-name Z, so that any error condition is obtained here via a GO TO 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. |

### *St*atus 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

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 example8**) produces the following display. Note that pressing a key will terminate the program.

```
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 use IE to 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 contents 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 contents 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 contents of a COBOL data item. Next, the document is transformed into another format (the same format as in "Example 2: Export File and Import File with Style Sheets" described on page 96) and then transformed back into the original output format. Then the program reads (or imports) the same XML document and places the contents 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 (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (page 62). The XML EXPORT FILE statement constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE (page 65). The XML IMPORT FILE statement reads an XML document (from a file) into a COBOL data item.
- XML TRANSFORM FILE (page 69). The XML TRANSFORM FILE statement uses a style sheet to modify (transform) an XML document into another format.
- XML TERMINATE (page 80). The XML TERMINATE statement 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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **example9.cob**. The K Option "kills" 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. It is possible for XML INITIALIZE to fail; therefore, 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 contents of the XML document are 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 contents of the XML document are imported from the file, **liant9c.xml**, and placed in the same data item using the XML IMPORT FILE statement.

Subsequently, the contents of the XML document, **liant9c.xml**, are transformed using the style sheet 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 contents of the data item defined in the copy file, **liant.cpy,** are 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 also 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 contents of this data item are 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 Toolkit 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 style sheet 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 style sheet 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 style sheet 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-name Z, so that any error condition is obtained here via a GO TO 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

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 example9**) produces the following display. Note that pressing a key will terminate the program.

```
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 use IE to 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 contents 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="2002-10-07T14:10:24">
  <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: Well Formed and Validate Diagnostic Messages

This program illustrates the diagnostic messages that may be displayed for XML documents that are not well formed or valid. The program used the XML TEST WELLFORMED-FILE and XML VALIDATE FILE statements to test and validate a series of XML documents.

This example uses the following XML statements:

- XML INITIALIZE (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML TEST WELLFORMED-FILE (page 67). The XML TEST WELLFORMED-FILE statement verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE FILE (page 70). The XML VALIDATE FILE statement verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML TERMINATE (page 80). The XML TERMINATE statement 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 Utility, **rmpgmcom**.

## 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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **exampleA.cob**. The K Option "kills" 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. It is possible for XML INITIALIZE to fail; therefore, the return status must be checked before continuing.

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 contents of the data item defined in the copy file, **liant.cpy**, are 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 also 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 contents of this data item are 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 Toolkit 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 "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 "XLiantA2" "ExampleA".` | Execute the XML VALIDATE FILE statement specifying: the XML document filename 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-name Z, so that any error condition is obtained here via a GO TO 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

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 after the first diagnostic message, the second is after the second diagnostic message, and the third is after some successful tests.

#### *First Display*

Note that pressing a key will cause the program to continue.

```
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 2002/10/\
   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:
```

### *Second Display*

Note that pressing a key will cause the program to continue.

```
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 2002/10/\
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:
```

### *Third Display*

Note that 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. A COBOL program and an XML document file may contain the same elementary items, but may not have the identical structure. The XML Toolkit 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. (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 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, the XML Toolkit 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 (page 80). The XML INITIALIZE statement initializes or opens a session with the **xmlif** library.
- XML EXPORT FILE (page 62). The XML EXPORT FILE statement constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE (page 65). The XML IMPORT FILE statement reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (page 80). The XML TERMINATE statement 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 Utility, **rmpgmcom**.

## 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.

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 Utility, **rmpgmcom**, which is shipped with the RM/COBOL development system, is used for this purpose.

Line 6 executes **exampleB.cob**. The K Option "kills" 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. It is possible for XML INITIALIZE to fail; therefore, 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 contents of the XML document are imported from the file, **LiantB.xml**, and placed in the same data item using the XML IMPORT FILE statement.

Additionally, the contents 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 contents of the data item defined in the copy file, **liant.cpy,** are 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 also 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 contents of this data item are 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 Toolkit 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-name Z, so that any error condition is obtained here via a GO TO 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

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 exampleB**) produces the following display. Note that pressing a key will terminate the program.

```
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 use IE to 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 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 contents of this file are 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" (see 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 contents of this file are similar to the following:

```
@echo off
call cleanup

@echo Example1 - Export / Import File.
call example example1 Liant-Address

@echo Example2 - Export / Import with style sheets.
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
```

This batch file has no parameters. Run it by entering the following on the command line:

```
Examples
```

# Appendix B: XML Toolkit Sample Application Programs

The XML Toolkit 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 the XML Toolkit within a realistic context so that you can better see how to integrate them into your own applications. This appendix describes how to use and access these sample application programs.

## Using the Sample Application Programs

The sample application programs are included in the XML Toolkit samples directory, **Samples**. As shipped from Liant, this directory contains only a single HTML file. Viewing this file with your Web browser will direct you to an XML Toolkit samples page on the Liant Web site at:

http://www.liant.com/xmltk/samples

This page contains a list of links to the various sample applications. Selecting a sample will cause that sample to be downloaded and installed on your computer. For example, selecting the Directory Split (DirSplit) sample will download this application in the **DirSplit** subdirectory of the **Samples** directory.

**Note** The most complete and up-to-date versions of the XML Toolkit sample programs can be found on the Liant Web site shown above.

# Appendix C: XML Toolkit Error Messages

This appendix lists and describes the messages that can be generated during the use of the XML Toolkit for RM/COBOL.

## Error Message Format

XML Toolkit 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** A table listing the error messages begins on page 197.

### 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.

The following illustrates an example of the first line of an error message:

```
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 2002/08/29 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 2002/08/29 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

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

***Table1: XML Toolkit for RM/COBOL Error Messages***

| Message Number | Message Text | Description |
|---|---|---|
| 0 | Success | A normal completion occurred, no informative message, warning or error was detected. |
| 1 | Informative- indicated 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. |

***Table 1: XML Toolkit for RM/COBOL Error Messages (Cont.)***

| Message Number | Message Text | Description |
|---|---|---|
| 10 | Error - cannot create URL | The Xmlif library detected that a URL (a string beginning with the sequence "http://") 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. |
| 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. |

***Table 1: XML Toolkit for RM/COBOL Error Messages (Cont.)***

| Message Number | Message Text | Description |
|---|---|---|
| 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 a style sheet or schema. |
| 25 | Error – invalid data address | The xmlif library has detected that the data structure address specified in an XML IMPORT or XML EXPORT statement 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 XML EXPORT statement 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. |

***Table 1: XML Toolkit for RM/COBOL Error Messages (Cont.)***

| Message Number | Message Text | Description |
|---|---|---|
| 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). 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 XML style sheet. 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. |
| 31 | Error - in function: LoadStyleSheetFromText | An error was detected while trying to load an XML style sheet. 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 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). 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 data name command-line parameter is missing. |
| 35 | Error - subscript out of range | The xmlif library while executing an XML IMPORT 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). |

***Table 1: XML Toolkit for RM/COBOL Error Messages (Cont.)***

| Message Number | Message Text | Description |
|---|---|---|
| 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) style sheet. This error may occur in the xmlif library. |
| 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 7.5 or newer runtime 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 | Wrong COBOL object version | The cobtoxml utility has determined that the COBOL object version being used is newer than was available when this XML Toolkit version was released and, therefore, may contain features that are not supported by the XML Toolkit. Check with Liant Software for updates to the XML Toolkit. |
| 44 | 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 the XML Toolkit is being used but that the model files were generated with an older cobtoxml utility. |

# Glossary of Terms

**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 the XML Toolkit, 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.

**XHTML.** Extensible HyperText Markup Language.

**Schema valid XML document.** An XML document that conforms to a particular XML schema.

**UNC.** Universal Naming Convention.

**URL.** Universal Resource Locator.

**Valid XML document.** See **Schema valid XML document**.

**Well-formed XML document.** An XML document 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.

**XML.** Extensible Markup Language.

**XML schema.** An XML document that specifies the structure and allowed content for another XML document.

**XSLT.** XML Style Sheet Language for Transformation.

# Index

## T



## U



## V



## W



## X