Liant Software Corporation

XML Extensions®

User's Guide

Second Edition

LIANT

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

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

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

Liant Software Corporation
5914 West Courtyard Dr., Suite 100
Austin, TX 78730-4911
U.S.A.

Phone (512) 343-1010
(800) 762-6265
Fax (512) 343-9487

Web site http://www.liant.com

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

Document Edition Number	Applies To Product Version	Publication Date
Second	XML Extensions version 12 and later	October 2008
First	XML Extensions version 9	January 2005

Important Liant documentation is provided in Adobe PDF. All manuals are distributed in Adobe Acrobat (.PDF) format and require the Adobe Acrobat Reader, 6.0 or higher, in order to display them. The installation program for the Adobe Acrobat Reader is available from the Adobe web site at: www.adobe.com.

Contents

Preface 1

Welcome to XML Extensions 1
About Your Documentation 2
Related Publications 3
Symbols and Conventions 3
Registration 4
Technical Support 5
    Support Guidelines 5
    Test Cases 5

Chapter 1: Installation and Introduction 7

Before You Start 7
    System Requirements 7
        For Windows 7
        For UNIX 8
    XML Extensions Components 8
        Development 8
        Deployment 9
Installing XML Extensions 9
    Installing on Windows 10
        Install the Development System on Windows 10
        Install the Deployment System on Windows 10
    Installing on UNIX 10
        Install the Development System on UNIX 10
        Install the Deployment System on UNIX 10
Introducing XML Extensions 11
    What is XML? 11
        COBOL as XML 12
        XML as COBOL 15

Chapter 2: Getting Started with XML Extensions 17

Overview 17
Typical Development Process Example 18
    Design the COBOL Data Structure and Program Logic 19
    Compile the Program 19
    Execute the COBOL Program 19
        Making a Program Skeleton 20
        Making a Program that Exports an XML Document 21
        Populating the XML Document with Data Values 22
    Deploy the Application 23
How XML Extensions Locates Files 23

Chapter 3: XML Extensions Statements Reference 25

What are XML Extensions Statements? 25
Memory Management with XML Extensions 26
Searching for Files 26
Document Processing Statements 27
XML EXPORT FILE 28
XML EXPORT TEXT 31
XML IMPORT FILE 34
XML IMPORT TEXT 36
XML TEST WELLFORMED-FILE 38
XML TEST WELLFORMED-TEXT 38
XML TRANSFORM FILE 39
XML VALIDATE FILE 40
XML VALIDATE TEXT 41
Document Management Statements 42
XML COBOL FILE-NAME 43
XML FREE TEXT 44
XML GET TEXT 45
XML PUT TEXT 46
XML REMOVE FILE 46
XML RESOLVE DOCUMENT-NAME 47
XML RESOLVE SCHEMA-FILE 47
XML RESOLVE STYLESHEET-FILE 48
XML RESOLVE MODEL-NAME 49
Directory Management Statements 50
XML FIND FILE 51
XML GET UNIQUEID 52
State Management Statements 53
XML COMPATIBILITY MODE 55
XML INITIALIZE 55
XML TERMINATE 56
XML DISABLE ALL-OCCURRENCES 56
XML ENABLE ALL-OCCURRENCES 57
XML DISABLE ATTRIBUTES 57
XML ENABLE ATTRIBUTES 58
XML DISABLE CACHE 58
XML ENABLE CACHE 59
XML FLUSH CACHE 59
XML GET FLAGS 60
XML TRACE 60
XML GET STATUS-TEXT 62
XML SET ENCODING 63
XML SET FLAGS 64
XML SET XSL-PARAMETERS 64
XML CLEAR XSL-PARAMETERS 65

Chapter 4: COBOL Considerations 67

File Management 67
Automatic Search for Files 67
File Naming Conventions 68
External XSLT Stylesheet File Naming Conventions 68
Other Input File Naming Conventions 68
Other Output File Naming Conventions 68

Data Conventions....69
Data Representation....69
COBOL and Character Encoding....70
FILLER Data Items....71
Missing Intermediate Parent Names....72
Unique Element Names....73
Unique Identifier....74
Sparse COBOL Records....75
Copy Files....75
Statement Definitions....75
REPLACE Statement Considerations....76
Displaying Status Information....77
Application Termination....77
Anonymous COBOL Data Structures....78
Limitations....78
Data Items (Data Structures)....78
Edited Data Items....79
Wide and Narrow Characters....79
Data Item Size....79
Data Naming....79
OCCURS Restrictions....80
Reading, Writing, and the Internet....80
Optimizations....80
Occurs Depending....80
Empty Occurrences....81
Cached XML Documents....81

Chapter 5: XML Considerations....83

XML and Character Encoding....83
Document Type Definition Support....84
XSLT Stylesheet Files....85
Handling Spaces and Whitespace in XML....86
Schema Files....87

Appendix A: XML Extensions Examples....89

Example 1: Export File and Import File....90
Development for Example 1....90
Batch File for Example 1....90
Program Description for Example 1....91
Data Item for Example 1....91
Other Definitions for Example 1....92
Program Structure for Example 1....92
Execution Results for Example 1....94
Example 2: Export File and Import File with XSLT Stylesheets....95
Development for Example 2....96
Batch File for Example 2....96
Program Description for Example 2....96
Data Item for Example 2....97
Other Definitions for Example 2....97
Program Structure for Example 2....98
XSLT Stylesheets for Example 2....100
Execution Results for Example 2....102

Example 3: Export File and Import File with OCCURS DEPENDING 103
Development for Example 3 103
Batch File for Example 3 103
Program Description for Example 3 104
Data Item for Example 3 104
Other Definitions for Example 3 105
Program Structure for Example 3 106
Execution Results for Example 3 108
Example 4: Export File and Import File with Sparse Arrays 109
Development for Example 4 110
Batch File for Example 4 110
Program Description for Example 4 110
Data Item for Example 4 111
Other Definitions for Example 4 111
Program Structure for Example 4 111
Execution Results for Example 4 114
Example 5: Export Text and Import Text 120
Development for Example 5 120
Batch File for Example 5 120
Program Description for Example 5 121
Data Item for Example 5 121
Other Definitions for Example 5 122
Program Structure for Example 5 122
Execution Results for Example 5 125
Example 6: Export File and Import File with Directory Polling 126
Development for Example 6 126
Batch File for Example 6 127
Program Description for Example 6 127
Data Item for Example 6 128
Other Definitions for Example 6 128
Program Structure for Example 6 129
Execution Results for Example 6 132
Example 7: Export File, Test Well-Formed File, and Validate File 134
Development for Example 7 134
Batch File for Example 7 135
Program Description for Example 7 135
Data Item for Example 7 136
Other Definitions for Example 7 136
Program Structure for Example 7 137
Execution Results for Example 7 139
Example 8: Export Text, Test Well-Formed Text, and Validate Text 140
Development for Example 8 140
Batch File for Example 8 141
Program Description for Example 8 141
Data Item for Example 8 142
Other Definitions for Example 8 142
Program Structure for Example 8 143
Execution Results for Example 8 146
Example 9: Export File, Transform File, and Import File 147
Development for Example 9 147
Batch File for Example 9 148
Program Description for Example 9 148
Data Item for Example 9 149
Other Definitions for Example 9 149
Program Structure for Example 9 150
Execution Results for Example 9 152

Example A: Diagnostic Messages....154
Development for Example A....155
Batch File for Example A....155
Program Description for Example A....156
Data Item for Example A....156
Other Definitions for Example A....157
Program Structure for Example A....157
Execution Results for Example A....160
Example B: Import File with Missing Intermediate Parent Names....162
Development for Example B....163
Batch File for Example B....163
Program Description for Example B....164
Data Item for Example B....164
Other Definitions for Example B....165
Program Structure for Example B....165
Execution Results for Example B....168
Example C: Export File with Document Prefix....170
Development for Example C....170
Batch File for Example C....170
Program Description for Example C....171
Data Item for Example C....171
Document Prefix for Example C....171
Other Definitions for Example C....172
Program Structure for Example C....172
Execution Results for Example C....174
Example Batch Files....175
cleanup.bat....175
example.bat....175
examples.bat....176

Appendix B: XML Extensions Sample Application Programs 177

Accessing the Sample Application Programs....177

Appendix C: XML Extensions Error Messages 179

Error Message Format....179
Message Text....179
COBOL Traceback Information....180
Filename or Data Item in Error....180
Parser Information....180
Summary of Error Messages....181

Appendix D: slicexsy Utility Reference 189

What is the slicexsy Utility? 189
Things to Consider Before Using slicexsy 190
Using the slicexsy Utility 190
File Naming Conventions 191
Model File Naming Conventions 191
Backward Compatibility 191
Command Line Interface 192
Command Line Options 195
Banner Options 195
Schema Options 195
Model Files 196
Template File 196
Internal XSLT Stylesheet File 197
Schema File 197
Referencing XML Model Files 198

Appendix E: Summary of Enhancements 199

Version 12 199
Version 9 201
Version 2 202
Version 1 203

Glossary of Terms 205

Terminology and Definitions 205

Index 211

List of Tables

Table 1: XML Extensions Error Messages 181

Preface

Welcome to XML Extensions

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

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

Version 12 of XML Extensions runs on Microsoft Windows 32-bit operating systems and selected UNIX platforms. It requires RM/COBOL version 12 or later. The new features for the most recent release of XML Extensions are described in Appendix E: *Summary of Enhancements* (on page 199). (Information on the significant enhancements in previous releases of XML Extensions is also included in this appendix.) Deficiencies that are version-specific or are discovered after publication are described in the README files contained on the delivered media.

Notes

- The term "Windows" in this document refers to Microsoft 32-bit Windows operating systems, including Microsoft Windows 2000, Windows XP, Windows Server 2003, Windows Vista, or Windows Server 2008, unless specifically stated otherwise.
- XML Extensions makes use of underlying XML parsers from other vendors. On Windows, Liant develops using a specific version and service pack of Microsoft's MSXML, which is included with your installation. Microsoft will occasionally ship updates to a given version and service pack that either enhance security or fix problems. You may monitor their web site for the latest updates, although it may not be advisable to update to a higher version or service pack. Check with Liant technical support to ensure that the update is necessary. On UNIX or Linux systems, XML Extensions links to specific libxml or libxslt libraries from the GNOME project for each release. It is not possible for the developer or end-user to upgrade these libraries.

About Your Documentation

XML Extensions documentation consists of a user's guide, which is distributed electronically in Portable Document Format (PDF) as part of the XML Extensions software distribution. It is also available on the Liant web site at http://www.liant.com/docs.

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

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

Chapter 1—Installation and Introduction describes the installation process and system requirements, and provides a general overview of XML Extensions.

Chapter 2—Getting Started with XML Extensions presents the basic concepts used in XML Extensions by creating an example XML-enabled application.

Chapter 3—XML Extensions Statements Reference describes the statements that are used by XML Extensions at runtime.

Chapter 4—COBOL Considerations provides information specific to using RM/COBOL when developing an XML-enabled application.

Chapter 5—XML Considerations provides information specific to using XML when using XML Extensions with RM/COBOL to develop an XML-enabled application.

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

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

Appendix C—XML Extensions Error Messages lists and describes the messages that can be generated during the use of XML Extensions.

Appendix D—slicexsy Utility Reference describes the optional **slicexsy** utility (**slicexsy.exe** on Windows and **slicexsy** on UNIX) that is provided for backward compatibility and for schema validation.

Appendix E—Summary of Enhancements provides an overview of the new features in the current release, and reviews the changes and enhancements that were added to earlier releases of XML Extensions.

The *XML Extensions User's Guide* also includes a glossary (on page 205) and an index (on page 211).

Related Publications

For additional information, refer to the following publications:

RM/COBOL User's Guide

RM/COBOL Language Reference Manual

RM/COBOL Syntax Summary Help File

Xcentrisity Business Information Server (BIS) User's Guide

Symbols and Conventions

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

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

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

   In COBOL syntax, italic text denotes a placeholder or variable for information you supply, as described below.
5. The symbols found in the COBOL syntax charts are used as follows:
   a. *italicized words* indicate items for which you substitute a specific value.
   b. UPPERCASE WORDS indicate items that you enter exactly as shown (although not necessarily in uppercase).
   c. ... indicates indefinite repetition of the last item.
   d. | separates alternatives (an either/or choice).
   e. [ ] enclose optional items or parameters.
   f. { } enclose a set of alternatives, one of which is required.
   g. {| |} surround a set of unique alternatives, one or more of which is required, but each alternative may be specified only once; when multiple alternatives are specified, they may be specified in any order.
6. All punctuation must appear exactly as shown.
7. Key combinations are connected by a plus sign (+), for example, Ctrl+X. This notation indicates that you press and hold down the first key while you press the second key. For

example, “press Ctrl+X” means to press and hold down the Ctrl key while pressing the X key. Then release both keys.

8. Note the distinction of the following terminology:

   - The term “window” refers to a delineated area of the screen, normally smaller than the full screen.
   - The term “Windows” in this document refers to 32-bit Microsoft Windows operating systems, including Microsoft Windows 2000, Windows XP, Windows Server 2003, Windows Vista, or Windows Server 2008, unless specifically stated otherwise.

     XML Extensions was never supported on Windows 95. Beginning with version 11, XML Extension no longer supports earlier Microsoft Windows operating systems, including Microsoft Windows 98, Windows 98 SE, Windows Me, and Windows NT 4.0.

9. RM/COBOL Compile and Runtime Command line options may be preceded by a hyphen. If any option on a command line is preceded by a hyphen, then a leading hyphen is required for all options. When assigning a value to an option, the equal sign is optional if leading hyphens are used.

Registration

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

Registering your product entitles you to the following benefits:

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

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

Technical Support

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

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

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

Support Guidelines

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

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

Test Cases

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

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

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

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

Chapter 1: Installation and Introduction

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

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

Before You Start

Before you follow the instructions for installing XML Extensions (on page 9), make sure that your computer configuration meets the following minimum hardware and software requirements for each of the supported architectures, and that your XML Extensions package contains the necessary components for development and deployment.

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

System Requirements

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

For Windows

The system requirements for Windows include the following:

- The XML Extensions hardware and software requirements are the same as RM/COBOL version 12 for 32-bit Windows. (See the *RM/COBOL User's Guide, Second Edition* or later.) Additionally, XML Extensions may be used in conjunction with Terminal Server.
- Microsoft's XML parser, MSXML 6.0 or greater, is also required. (A schema processor and an XSLT transformation processor are included in the Microsoft MSXML 6.0 parser.)

Note The MSXML 6.0 parser may fail to install correctly if the target system does not have either Microsoft Windows Installer or Internet Explorer installed. Both of these products are freely available from Microsoft. To obtain these applications, follow the www.microsoft.com/downloads/search.asp link and search for the keywords "Windows Installer 2.0" or "Internet Explorer", as needed.

For UNIX

The system requirements for UNIX include the following:

- The XML Extensions hardware and software requirements are the same as RM/COBOL version 12 for UNIX. (See the *RM/COBOL User's Guide, Second Edition* or later.)

  Notes

  - The XML parser (libxml) and the XSLT transformation processor (libxslt) from the C libraries for the Gnome project are included in XML Extensions.
  - While the Windows implementation continues to support the use of schema files (on page 87), the UNIX implementation does not currently support this capability. Schema support in the underlying XML parser (libxml) is still under development.

XML Extensions Components

The XML Extensions package contains the following components for development and deployment.

Development

The XML Extensions development system includes the following:

- The RM/COBOL compiler, which has been specifically licensed for XML Extensions.
- Deployment files. These files are listed in Deployment (on page 9).
- Copy files (**lixmlall.cpy**, **lixmldef.cpy**, **lixmldsp.cpy**, **lixmlrpl.cpy**, and **lixmltrm.cpy**). For more details, see Copy Files (on page 75).
- Example files. These programs or program fragments illustrate how XML Extensions statements are used. For further information, see Appendix A: *XML Extensions Examples* (on page 89). The example programs can be found in the XML Extensions example directory, **Examples**.
- Sample files. These self-contained, working application programs, which include the complete source, can be used in your own applications by modifying or customizing them, as necessary. See Appendix B: *XML Extensions Sample Application Programs* (on page 177) for more details. The sample application programs can be found in the XML Extensions sample directory, **Samples**.
- **slicexsy** command line utility (**slicexsy.exe** on Windows and **slicexsy** on UNIX). For more information, see Appendix D: *slicexsy Utility Reference* (on page 189).
- XML document files used by the **slicexsy** utility (**scstrict.xsl** and **toxsln.xsl**).

Deployment

The XML Extensions deployment system consists of the following files:

- **xmlif** COBOL-callable subprogram library (**xmlif.dll** on Windows and **xmlif.so** on UNIX). For more information, see Chapter 3: *XML Extensions Statements Reference* (on page 25).
- For Windows, MSXML 6.0, the Microsoft XML parser, schema processor, and XSLT transformation processor (**msxml6.dll**, **msxml4a.dll**, and **msxml4r.dll**).

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

Note If the **slicexsy** utility is used to generate model files, then at least the template file (**.xtl** extension) must be deployed when the XML symbol table is omitted from the deployed COBOL object file. If schema validation is to be performed, then all three model files (.**xtl**, **.xsl**, and **.xsd**) must be deployed along with the COBOL object files. Normally, model files are stored in the same location as the COBOL object files. For more information, see Appendix D: *slicexsy Utility Reference* (on page 189).

Installing XML Extensions

The following sections describe the distribution media options, and how to install the XML Extensions development and deployment systems on Windows (on page 10) and UNIX (on page 10). XML Extensions is available as a development system and a deployment system. The development system is designed to operate in conjunction with an RM/COBOL development system and the software is provided with the RM/COBOL development system. The deployment system is designed to operate in conjunction with an RM/COBOL runtime system and the software is delivered with the RM/COBOL runtime system.

For development, both the XML Extensions development system and Liant's RM/COBOL version 12 development system are required. In addition, an XML Extensions license is required for development.

For deployment, both the XML Extensions deployment system and the RM/COBOL version 12 runtime system are required. The XML Extensions deployment system does not require an additional license beyond the license requirements of the RM/COBOL runtime system. However, the XML Extensions deployment system is only useful for an application that has been created using the XML Extensions development system, which is licensed.

Installing on Windows

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

Install the Development System on Windows

The XML Extensions development system is included with the RM/COBOL development system, but requires a separate license. The license certificate file is normally obtained electronically in an email. Once obtained, the license is installed using the **licverifyall.exe** utility that is provided with the RM/COBOL development system. The license should be installed into the same **license.vlt** file that contains the RM/COBOL development system, that is, the **license.vlt** file in the installation directory of the RM/COBOL development system.

Install the Deployment System on Windows

The XML Extensions deployment system is included with the RM/COBOL runtime system. The end user of the application must have a licensed RM/COBOL runtime system installed, but does not need any additional license or software to use an application developed with the XML Extensions development system.

Installing on UNIX

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

Install the Development System on UNIX

The XML Extensions development system is included with the RM/COBOL development system, but requires a separate license. The license certificate file is normally obtained electronically in an email. Once obtained, the license is installed using the **licverifyall** utility that is provided with the RM/COBOL development system. The license should be installed into the same **license.vlt** file that contains the RM/COBOL development system, that is, the **license.vlt** file in the installation directory of the RM/COBOL development system.

For additional information on how to access the license certificate file when, for example, it is provided on a diskette, see Chapter 2: *Installation and System Considerations for UNIX* of the *RM/COBOL User's Guide, Second Edition*.

Install the Deployment System on UNIX

The XML Extensions deployment system is included with the RM/COBOL runtime system. The end user of the application must have a licensed RM/COBOL runtime system installed, but does not need any additional license or software to use an application developed with the XML Extensions development system

Introducing XML Extensions

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

Note A COBOL data structure, as used in this document, 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. An XML Extensions-enabled RM/COBOL compiler generates and embeds an XML-format symbol table in the COBOL object file. The XML-format symbol table provides a map between the COBOL data structure specified in an XML Extensions statement and the XML representation of the COBOL data structure. This map can be used to move data in either direction at runtime. Extensible Stylesheet Language Transformations (XSLT) of the XML data representation can be used to match XML element names to COBOL data-names in cases where the names differ.

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

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

In order to add this powerful document-handling capability to a COBOL application, the programmer need only describe the information to be received or transmitted to the external components as COBOL data definitions. In many cases, this description will simply be the already-existing data area defined in the COBOL application.

What is XML?

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

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

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

COBOL as XML

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

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

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

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

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

This is not a new problem; it is one that COBOL programmers (as well as other application programmers) have had to deal with on an ad hoc basis since the beginning of the computer age. But now, XML gives us a way to encode all of the information in a generally understandable way.

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

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

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

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

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

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

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

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

You do this with the help of an important XML tool called an external XSLT stylesheet (see page 85). (In this document, “external XSLT stylesheet” is used to differentiate an XSLT stylesheet provided by the user from the “internal XSLT stylesheet” generated as one of the model files by the optional **slicexsy** utility.) For now, assume that an XSLT stylesheet can transform an XML document into any desired alternative XML document. If this is true (and it is), you must code the incoming attributes as something that has a direct COBOL counterpart. This would be as a data item represented as a text element in XML.

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

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

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

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

Chapter 2: Getting Started with XML Extensions

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

Overview

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

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

XML Extensions consists of the following main components:

- The RM/COBOL compiler, which has been specifically licensed for XML Extensions.
- A COBOL-callable runtime library (**xmlif.dll** on Windows and **xmlif.so** on UNIX). This library is used to implement a set of XML Extensions statements specified in a COBOL program that are available to the developer for directing the importing and exporting of COBOL data as XML. For more information, see Chapter 3: *XML Extensions Statements Reference* (on page 25).
- The optional **slicexsy** utility (**slicexsy.exe** on Windows and **slicexsy** on UNIX), which runs as a post-compile step. This program creates a set of XML documents, called model files (on page 196), which describe a selected COBOL data structure as a set of XML documents.

Typical Development Process Example

This section provides an example of how to produce an XML-enabled application. These instructions assume that both the XML Extensions development system and the RM/COBOL development system (version 12 or later) are installed on your computer.

Note More examples and information about complete sample application programs can be found in Appendix A: *XML Extensions Examples* (on page 89), Appendix B: *XML Extensions Sample Application Programs* (on page 177), and in the XML Extensions examples and samples directories (**Examples** and **Samples**, respectively).

The basic steps to developing an XML-enabled application are as follows:

1. Design the COBOL data structure and program logic (see page 19). Develop a COBOL program, or modify an existing one, using XML Extensions statements.
2. Compile the program (see page 19). Use an RM/COBOL compiler that is licensed for XML Extensions with the configuration file option, COMPILER-OPTIONS SUPPRESS-XML-SYMBOL-TABLE, set to a value of NO, which, by default, results in the production of the XML-format symbol table.

   Note On the development machine, a large XML-format symbol table may necessitate an increase in system resources, including the addition of hardware (for example, memory and/or disc space) or system configuration modifications.
3. Execute the COBOL program (see page 19). Test the program and repeat steps 1 and 2, as necessary.
4. Deploy the application (see page 23). Distribute the XML Extensions deployable files. These files consist of the **xmlif** library and the underlying XML parser that this library uses.

Note As an alternative to specifying the data structure name using the *ModelFileName#DataFileName* parameter of the XML IMPORT FILE statement (see page 34), you can run the optional slicexsy utility (see page 189) in order to select a portion, or "slice," of the XML-format symbol table that contains a single data structure. The **slicexsy** utility generates a set of XML model files that describe a data structure within the COBOL program. If the **slicexsy** utility generates a schema file, it also generates a stylesheet. Both should be deployed with the application.

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

Design the COBOL Data Structure and Program Logic

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

The generation of an XML-format symbol table is controlled by whether or not the RM/COBOL compiler is licensed for XML Extensions and also by the following configuration file option:

```
COMPILER-OPTIONS SUPPRESS-XML-SYMBOL-TABLE=<value>
```

where, <*value*> may be YES or NO. The default value of the SUPPRESS-XML-SYMBOL-TABLE keyword is NO, resulting in the production of an embedded XML-format symbol table by default when the RM/COBOL compiler is licensed for XML Extensions.

Compile the program with the following command line:

```
rmcobol getstarted
```

Execute the COBOL Program

Next, you execute and test the program.

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

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

Making a Program Skeleton

Step 1 started with just a fragment of the program in order to show the COBOL data structure.

The interface to the **xmlif** library, a COBOL-callable subprogram, is simplified by using some COBOL copy files that perform source text replacement. This means that the developer may use XML Extensions statements, 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 Extensions statements. The copy file, **lixmlall.cpy** (or at least the copy files referenced by **lixmlall.cpy**), must be copied in the Working-Storage Section of the program in order to use XML Extensions. For more information, see Copy Files (on page 75).

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

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

< insert COBOL PROCEDURE DIVISION logic here >

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

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

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

```
rmcobol  getstarted
runcobol getstarted
```

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

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

Making a Program that Exports an XML Document

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

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

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

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

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

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

As you would expect, the data structure name is `customer-address`. 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
runcobol getstarted
```

Populating the XML Document with Data Values

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

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

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

     XML EXPORT FILE
         Customer-Address
         "Address"
         "getstarted#customer-address".
     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 is used to provide content for the data structure.

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

```
rmcobol  getstarted
runcobol getstarted
```

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

```
<?xml version="1.0" encoding="UTF-8" ?>
 <customer-address xmlns:xtk="http://liant.com/xcentrisity/
     \xml-extensions/symbol-table/">
  <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>
```

Deploy the Application

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

Deploy the **xmlif** library and the underlying XML parser that it uses. If you chose to run the **slicexsy** utility, deploy the model files that it generates. Normally, these files are stored in the same location as the COBOL program files.

How XML Extensions Locates Files

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

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

  On Windows

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

  On UNIX

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

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

  On Windows

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

  On UNIX

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

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

  On Windows

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

  On UNIX

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

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

For additional information on how XML Extensions locates files, see the following:

- Automatic Search for Files (on page 67)
- File Naming Conventions (on page 68)
- UNIX Character Encoding (on page 70)
- Windows Character Encoding (on page 70)

Chapter 3: XML Extensions Statements Reference

This chapter describes the statements that are used by XML Extensions at runtime.

What are XML Extensions Statements?

XML Extensions statements allow you to process, manipulate, and validate XML documents. The statements are contained in the 32-bit dynamic link library on Windows (**xmlif.dll**) or the shared object on UNIX (**xmlif.so**) that is callable from RM/COBOL object programs.

On Windows, XML Extensions statements use the Microsoft MSXML 6.0 parser; on UNIX, XML Extensions statements use the XML parser (libxml) and the XSLT transformation processor (libxslt) from the C libraries for the Gnome project. For additional information, see Installing XML Extensions (on page 9) and the "Deployment" section in XML Extensions Components (on page 8).

XML Extensions statements are grouped into the following categories:

- Document Processing Statements (on page 27). These statements are used to process, manipulate, or validate XML documents.
- Document Management Statements (on page 42). These statements are used to copy an XML document from an external file to an internal text string and vice versa.
- Directory Management Statements (on page 50). These statements are useful when implementing directory-polling schemes.
- State Management Statements (on page 53). These statements are used to control the state or condition of XML Extensions statements.

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, flags, and any model files produced by the optional **slicexsy** utility (see Appendix D: *slicexsy Utility Reference* on page 189). In some statements, trailing positional parameters are optional and may be omitted, as specified in the statement descriptions in this chapter.

Memory Management with XML Extensions

At execution time, XML Extensions allocates memory and caches stylesheets and other artifacts of the XML document handling process. This is a standard technique to enhance performance, trading reduced execution time for additional memory usage. However, it is possible for a long running program that processes a substantial number of different XML documents to cause enough additional memory allocation that performance degrades, typically due to virtual memory swapping. As an example, a program might sit in a loop, waiting for an XML document to arrive in a directory; see the example for XML FIND FILE (on page 51).

The program may use the XML TERMINATE statement to cause all memory allocated by XML Extensions (with the exception of the document returned by the XML GET TEXT and XML IMPORT TEXT statements) for the run unit to be released. However, the XML INITIALIZE statement and any other XML Extensions statements that control optional behavior (for example, XML ENABLE ALL-OCCURRENCES) must be called to re-establish the XML environment before additional XML documents are processed.

Searching for Files

Model files are the XML documents generated by the optional **slicexsy** utility. XML Extensions uses model files only as input files. When XML Extensions references a model file, the appropriate predetermined extension is added, regardless of the presence or lack of an extension on the model file parameter supplied by the COBOL program. For more information, see Referencing XML Model Files (on page 198).

XML Extensions 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://, https://, or "file://").

Document Processing Statements

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

- **Export statements.** XML EXPORT FILE (on page 28) and XML EXPORT TEXT (on page 31) 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 (on page 34) and XML IMPORT TEXT (on page 36) are available to convert the content of an XML document—either an external file or an internal text string—to a COBOL data item.
- T**est and validation statements.** XML TEST WELLFORMED-FILE (on page 38), XML TEST WELLFORMED-TEXT (on page 38), XML VALIDATE FILE (on page 40), and XML VALIDATE TEXT (on page 41) are available to verify that an XML document—either an external file or an internal text string—is well-formed or valid.
- **Transformation statement.** Lastly, XML TRANSFORM FILE (on page 39) transforms an XML document in an external file into a new external file by applying an XSLT stylesheet. The resulting file may have almost any form, including XML, HTML, PDF, RTF, and so forth.

XML EXPORT FILE

This statement has the following parameters:

Parameter	Description
DataItem	The name of the COBOL data item that contains data to be exported.
DocumentName	The name of a file that will receive the exported XML document.
ModelFileName#DataFileName	This parameter may be either of the following: • The name of the data structure. If the # (hash) character is missing, it is assumed that the hash character was intended to be placed at the beginning of the parameter (as the specified name is a data item name). Previously, it was assumed that the # character was placed at the end of the parameter (as the specified name was a filename). Everything to the left of the "#" character is the filename. Everything to the right is the data structure name. Either component is optional; that is, model names of "file#", "file", "#data", "#" and "" are allowed. If the filename is omitted and a default has not been provided from the XML COBOL FILE-NAME statement (see page 43), then the current program is assumed. If the data name is not found, then calling programs are used (in order of the call stack). If the data name is omitted, then the entire program is used. If a hash "#" character is missing from the *ModelFileName* parameter, one will be assumed to be present at the beginning (the *ModelFileName* is assumed to be a data name). This is the default action. The default my be overridden by setting the RM_MISSING_HASH environment variable to either "trailing" or "file" to indicate a filename is present. The default may be explicitly specified by setting the RM_MISSING_HASH environment variable to either "leading" or "data" to indicate that a data name is present. Furthermore, a hierarchical specification of data names may be used; that is, "file#a//b//c" is valid. It has the same meaning as the COBOL specification of "C of B of A". Either data names or program names may be specified in the data name hierarchy; that is, A and B could be the names of programs, assuming B is a program contained in A. • The name of the set of XML files produced by the slicexsy utility that describe the COBOL data item. For more information, see Model Files (on page 196).
[*StyleSheetName*]	Optional. The name of an external XSLT stylesheet that will be used to transform the generated XML document before it is stored.
[*DocumentPrefix*]	Optional. A literal or the name of a COBOL data item that contains a document prefix; for example, a document type definition (DTD), which is to be output between the XML header and the first element of the exported XML document. For more information, see Document Type Definition Support (on page 84).

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#DataFileName* parameter. The output of this conversion is to the file specified by the *DocumentName* parameter. If the optional *StyleSheetName* parameter is present, the external XSLT stylesheet is used to transform the document after it has been generated but before it is stored in the data file.

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

Examples

Without an External XSLT Stylesheet:

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

With an External XSLT Stylesheet:

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

Without an External XSLT Stylesheet and With a Document Prefix:

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

With an External XSLT Stylesheet and Parameters:

```
XML SET XSL-PARAMETERS
     "MY-COUNT", 7.
IF NOT XML-OK GO TO Z.
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#DataFileName	This parameter may be either of the following: • The name of the data structure. If the # (hash) character is missing, it is assumed that the hash character was intended to be placed at the beginning of the parameter (as the specified name is a data item name). Previously, it was assumed that the # character was placed at the end of the parameter (as the specified name was a filename). Everything to the left of the "#" character is the filename. Everything to the right is the data structure name. Either component is optional; that is, model names of "file#", "file", "#data", "#" and "" are allowed. If the filename is omitted and a default has not been provided from the XML COBOL FILE-NAME statement (see page 43), then the current program is assumed. If the data name is not found, then calling programs are used (in order of the call stack). If the data name is omitted, then the entire program is used. If a hash "#" character is missing from the *ModelFileName* parameter, one will be assumed to be present at the beginning (the *ModelFileName* is assumed to be a data name). This is the default action. The default my be overridden by setting the RM_MISSING_HASH environment variable to either "trailing" or "file" to indicate a filename is present. The default may be explicitly specified by setting the RM_MISSING_HASH environment variable to either "leading" or "data" to indicate that a data name is present. Furthermore, a hierarchical specification of data names may be used; that is, "file#a//b//c" is valid. It has the same meaning as the COBOL specification of "C of B of A". Either data names or program names may be specified in the data name hierarchy; that is, A and B could be the names of programs, assuming B is a program contained in A. • The name of the set of XML files produced by the **slicexsy** utility that describe the COBOL data item. For more information, see Model Files (on page 196).
[*StyleSheetName]*	Optional. The name of an external XSLT stylesheet that will be used to transform the generated XML document before it is stored.
[*DocumentPrefix*]	Optional. A literal or the name of a COBOL data item that contains a document prefix, for example, a document type definition (DTD), which is to be output between the XML header and the first element of the exported XML document. For more information, see Document Type Definition Support (on page 84).

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#DataFileName* parameter, and then it is output as a text string. The address of the text string is placed in the COBOL pointer data item specified by the *DocumentPointer* parameter. If the optional *StyleSheetName* parameter is present, the external XSLT stylesheet is used to transform the document after it has been generated but before it is stored as a text string.

A block of memory is allocated to hold the generated XML document. The descriptor of this memory block overrides any existing address descriptor in the COBOL pointer data item. The COBOL application is responsible for releasing this memory when it is no longer needed by using XML FREE TEXT (see page 44).

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

Examples

Without an External XSLT Stylesheet:

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

With an External XSLT Stylesheet:

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

Without an External XSLT Stylesheet and With a Document Prefix:

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

With an External XSLT Stylesheet and Parameters:

```
XML SET XSL-PARAMETERS
     "MY-COUNT", 7.
IF NOT XML-OK GO TO Z.
XML EXPORT TEXT
    MY-DATA-ITEM
    "MY-DOCUMENT.XML"
    "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#DataFileName	This parameter may be either of the following: • The name of the data structure. If the # (hash) character is missing, it is assumed that the hash character was intended to be placed at the beginning of the parameter (as the specified name is a data item name). Previously, it was assumed that the # character was placed at the end of the parameter (as the specified name was a filename. Everything to the left of the "#" character is the filename. Everything to the right is the data structure name. Either component is optional; that is, model names of "file#", "file", "#data", "#" and "" are allowed. If the filename is omitted and a default has not been provided from the XML COBOL FILE-NAME statement (on page 43), then the current program is assumed. If the data name is not found, then calling programs are used (in order of the call stack). If the data name is omitted, then the entire program is used. If a hash "#" character is missing from the *ModelFileName* parameter, one will be assumed to be present at the beginning (the *ModelFileName* is assumed to be a data name). This is the default action. The default my be overridden by setting the RM_MISSING_HASH environment variable to either "trailing" or "file" to indicate a filename is present. The default may be explicitly specified by setting the RM_MISSING_HASH environment variable to either "leading" or "data" to indicate that a data name is present. Furthermore, a hierarchical specification of data names may be used; that is, "file#a//b//c" is valid. It has the same meaning as the COBOL specification of "C of B of A". Either data names or program names may be specified in the data name hierarchy; that is, A and B could be the names of programs, assuming B is a program contained in A. • The name of the set of XML files produced by the slicexsy utility that describe the COBOL data item. For more information, see Model Files (on page 196).
[*StyleSheetName*]	Optional. The name of an external XSLT stylesheet that will be used to transform the imported XML document before it is stored in the data item.

Description

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

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

Examples

Without an External XSLT Stylesheet:

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

With an External XSLT Stylesheet:

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

With an External XSLT Stylesheet and Parameters:

```
XML SET XSL-PARAMETERS
     "MY-COUNT", 7.
IF NOT XML-OK GO TO Z.
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#DataFileName	This parameter may be either of the following: • The name of the data structure. If the # (hash) character is missing, it is assumed that the hash character was intended to be placed at the beginning of the parameter (as the specified name is a data item name). Previously, it was assumed that the # character was placed at the end of the parameter (as the specified name was a filename. Everything to the left of the "#" character is the filename. Everything to the right is the data structure name. Either component is optional; that is, model names of "file#", "file", "#data", "#" and "" are allowed. If the filename is omitted and a default has not been provided from the XML COBOL FILE-NAME statement (on page 43), then the current program is assumed. If the data name is not found, then calling programs are used (in order of the call stack). If the data name is omitted, then the entire program is used. If a hash "#" character is missing from the *ModelFileName* parameter, one will be assumed to be present at the beginning (the *ModelFileName* is assumed to be a data name). This is the default action. The default my be overridden by setting the RM_MISSING_HASH environment variable to either "trailing" or "file" to indicate a filename is present. The default may be explicitly specified by setting the RM_MISSING_HASH environment variable to either "leading" or "data" to indicate that a data name is present. Furthermore, a hierarchical specification of data names may be used; that is, "file#a//b//c" is valid. It has the same meaning as the COBOL specification of "C of B of A". Either data names or program names may be specified in the data name hierarchy; that is, A and B could be the names of programs, assuming B is a program contained in A. • The name of the set of XML files produced by the slicexsy utility that describe the COBOL data item. For more information, see Model Files (on page 196).
[*StyleSheetName*]	Optional. The name of an external XSLT stylesheet that will be used to transform the imported XML document before it is stored in the data item.

Description

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

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

Examples

Without an External XSLT Stylesheet:

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

With an External XSLT Stylesheet:

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

With an External XSLT Stylesheet and Parameters:

```
XML SET XSL-PARAMETERS
     "MY-COUNT", 7.
IF NOT XML-OK GO TO Z.
XML IMPORT TEXT
    MY-DATA-ITEM
    "MY-DOCUMENT.XML"
    "MY-MODEL-FILE"
    "MY-STYLE-SHEET"
IF NOT XML-OK GO TO Z.
```

XML TEST WELLFORMED-FILE

This statement has the following parameter:

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

Description

The XML TEST WELLFORMED-FILE statement tests the XML document specified by the *DocumentName* parameter to see if it is well-formed. A well-formed XML document is one that conforms to XML syntax rules, but is not necessarily valid with respect to any schema. See XML VALIDATE FILE (on page 40) and XML VALIDATE TEXT (on page 41) for testing whether a document is valid with respect to a schema.

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

Example

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

XML TEST WELLFORMED-TEXT

This statement has the following parameter:

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

Description

The XML TEST WELLFORMED-TEXT statement tests the XML document specified by the *DocumentPointer* parameter to see if it is well-formed. A well-formed XML document is one that conforms to XML syntax rules, but is not necessarily valid with respect to any schema. See XML VALIDATE FILE (on page 40) and XML VALIDATE TEXT (on page 41) for testing whether a document is valid with respect to a schema.

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

Example

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

XML TRANSFORM FILE

This statement has the following parameters:

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

Description

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

Note Specifying the internal XSLT stylesheet file (one of the model files, discussed on page 196, created by the optional **slicexsy** utility) specified for the *StyleSheetName* parameter can be used to test the internal XSLT stylesheet transform, which is occasionally helpful in debugging problems with importing documents into COBOL.

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

Examples

With an External XSLT Stylesheet:

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

With an External XSLT Stylesheet and Parameters:

```
XML SET XSL-PARAMETERS
     "MY-COUNT", 7.
IF NOT XML-OK GO TO Z.
XML TRANSFORM FILE
    MY-DATA-ITEM
    "MY-DOCUMENT.XML"
    "MY-MODEL-FILE"
    "MY-STYLE-SHEET"
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/ModelFileName	The name of the schema file or set of model files that will be used to validate the XML document specified in *DocumentName*. **Note** If the **slicexsy** utility is used, *SchemaName* refers to a model filename. The template file produced by **slicexsy** (*modelfilename.xtl*) is parsed to determine the version of the model files. If the version is 12 or later, XML Extensions uses a two-step validation process: 1) *DocumentName* is transformed using a stylesheet (*modelfilename.xsl*); and 2) the schema validation is performed. If the version is prior to 12, XML Extensions performs validation using *SchemaName*.

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 is one that is both well-formed and has content that conforms to rules specified by an XML schema file. This schema file may be any of the following:

- one created using the optional **slicexsy** utility, as described in Appendix D: *slicexsy Utility Reference* (on page 189);
- one created by the **cobtoxml** utility used in XML Extensions prior to version 12; or
- one supplied by the user.

Note On UNIX systems, the underlying XML parser, **libxml**, does not support schema validation. The XML VALIDATE FILE statement on UNIX systems does not validate the XML document but does verify that it is well-formed.

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

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

Example

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

XML VALIDATE TEXT

This statement has the following parameters:

Parameter	Description
DocumentPointer	The name of a COBOL pointer data item that points to an XML document that is stored in memory as a text string.
SchemaName/ModelFileName	The name of the schema file or set of model files that will be used to validate the XML document specified in *DocumentPointer*. **Note** If the **slicexsy** utility is used, *SchemaName* refers to a model filename. The template file produced by **slicexsy** (*modelfilename.xtl*) is parsed to determine the version of the model files. If the version is 12 or later, XML Extensions uses a two-step validation process: 1) *DocumentPointer* is transformed using a stylesheet (*modelfilename.xsl*); and 2) the schema validation is performed. If the version is prior to 12, XML performs the validation using *SchemaName*.

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 is one that is both well-formed and has content that conforms to rules specified by an XML schema file. This schema file may be any of the following:

- one created using the optional **slicexsy** utility, as described in Appendix D: *slicexsy Utility Reference* (on page 189);
- one created by the **cobtoxml** utility used in XML Extensions prior to version 12; or
- one supplied by the user.

Note On UNIX systems, the underlying XML parser, **libxml**, does not support schema validation. The XML VALIDATE TEXT statement on UNIX systems does not validate the XML document but does verify that it is well-formed.

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

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

Example

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

Document Management Statements

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

- XML COBOL FILE-NAME (on page 43)
- XML FREE TEXT (on page 44)
- XML GET TEXT (on page 45)
- XML PUT TEXT (on page 46)
- XML REMOVE FILE (on page 46)
- A set of RESOLVE statements allows the developer to obtain a fully resolved pathname (for example, *c:\mystuff\stuff.xml* rather than *stuff.xml*), thus providing a globally unique name that can be passed as a parameter to a called sub-program. This is useful in cases where global resources are defined in the top-level program and then referenced in a called (possibly nested) subprogram that may include another resource having the same name. The RESOLVE statements (on page 47) include the following:
  - XML RESOLVE DOCUMENT-NAME
  - XML RESOLVE MODEL-NAME
  - XML RESOLVE STYLESHEET-NAME
  - XML RESOLVE SCHEMA-NAME

XML COBOL FILE-NAME

This statement has the following parameter:

Parameter	Description
[filename]	An optional string value that specifies the default *ModelFileName* value (the string before the #) in the *ModelFileName#DataFileName* parameter for subsequent statements that do not explicitly specify a *ModelFileName*. If omitted or specified with a value of spaces, the default *ModelFileName* value is reset to spaces, eliminating any previously set default *ModelFileName* value. If the parameter value is #, the name of the COBOL object file for the currently running COBOL program is used to set the default *ModelFileName* value. Otherwise, the current value of the parameter is used "as is" to set the default *ModelFileName* value.

Description

The XML COBOL FILE-NAME statement allows the developer to set the default *ModelFileName* (the string before the #) in the *ModelFileName#DataFileName* parameter of various subsequent XML Extensions statements. The default value will be used when the *ModelFileName* string is not specified in the *ModelFileName#DataFileName* parameter of those subsequent statements.

Example

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

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, which should have a value that has been set by the XML EXPORT TEXT statement (see page 31) or the XML GET TEXT statement (see page 45).

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 44) 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.
```

XML RESOLVE DOCUMENT-NAME

This statement has the following parameter:

Parameter	Description
DocumentName	The name of the XML document to be resolved.

Description

The XML RESOLVE DOCUMENT-NAME statement is used to resolve the name of an XML document file. The resolution process is the same as that for the *DocumentName* parameter of an XML IMPORT statement.

If the name is a URL, it is used "as is." Otherwise, if the name does not contain an extension, the extension **.xml** is added. If the file does not exist using the name as entered, then the RUNPATH environment variable is used to search for the file.

Example

```
XML RESOLVE DOCUMENT-NAME
            MY-DOCUMENT.
IF NOT XML-OK GO TO Z.
```

XML RESOLVE SCHEMA-FILE

This statement has the following parameter:

Parameter	Description
SchemaFileName	The name of the XML schema file to be resolved.

Description

The XML RESOLVE SCHEMA-FILE statement is used to resolve the name of an XML schema file (one of the model files, discussed on page 196, created using the optional **slicexsy** utility) specified for the *SchemaFileName* parameter. The resolution process is similar to that for the *ModelFileName#DataFileName* parameter of an XML IMPORT FILE, XML IMPORT TEXT, XML EXPORT FILE, or XML EXPORT TEXT statement. The value of this parameter must specify an existing template file (**.xtl** extension) and not a COBOL object file (**.cob** extension).

XML Extensions uses the model files only as input files. When XML Extensions references a model file, the appropriate predetermined extension is added, regardless of the presence or lack of an extension on the model file parameter supplied by the COBOL program. For more information, see Referencing XML Model Files (on page 198).

XML Extensions 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://", https://, or "file://"). If the name is a URL, it is used "as is." Otherwise, the file extension is forced to be **.xsd**. If the name does not exist, then the RUNPATH environment variable is used to search for the file.

Example

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

XML RESOLVE STYLESHEET-FILE

This statement has the following parameter:

Parameter	Description
StyleSheetName	The name of the XML stylesheet to be resolved.

Description

The XML RESOLVE STYLESHEET-FILE statement is used to resolve the name of an XML stylesheet file. The resolution process is the same as that for the *StyleSheetName* parameter of an XML IMPORT or XML EXPORT statement.

If the name is a URL, it is used "as is." Otherwise, if the name does not contain an extension, the extension **.xsl** is added. If the file does not exist using the name as entered, then the RUNPATH environment variable is used to search for the file.

Example

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

XML RESOLVE MODEL-NAME

This statement has the following parameter:

Parameter	Description
ModelFileName#DataFileName	This parameter may be either of the following: • The name of the data structure. If the # (hash) character is missing, it is assumed that the hash character was intended to be placed at the beginning of the parameter (as the specified name is a data item name). Previously, it was assumed that the # character was placed at the end of the parameter (as the specified name was a filename. Everything to the left of the "#" character is the filename. Everything to the right is the data structure name. Either component is optional; that is, model names of "file#", "file", "#data", "#" and "" are allowed. If the filename is omitted and a default has not been provided from the XML COBOL FILE-NAME statement (see page 43), then the current program is assumed. If the data name is not found, then calling programs are used (in order of the call stack). If the data name is omitted, then the entire program is used. If a hash "#" character is missing from the *ModelFileName* parameter, one will be assumed to be present at the beginning (the *ModelFileName* is assumed to be a data name). This is the default action. The default my be overridden by setting the RM_MISSING_HASH environment variable to either "trailing" or "file" to indicate a filename is present. The default may be explicitly specified by setting the RM_MISSING_HASH environment variable to either "leading" or "data" to indicate that a data name is present. Furthermore, a hierarchical specification of data names may be used; that is, "file#a//b//c" is valid. It has the same meaning as the COBOL specification of "C of B of A". Either data names or program names may be specified in the data name hierarchy; that is, A and B could be the names of programs, assuming B is a program contained in A. • The name of the set of XML files produced by the slicexsy utility that describe the COBOL data item. For more information, see Model Files (on page 196).

Description

The XML RESOLVE MODEL-NAME statement is used to resolve the name of a model file/data name combination. The resolution process is the same as that for the *ModelFileName#DataFileName* parameter of the XML IMPORT FILE, XML IMPORT TEXT, XML EXPORT FILE, or XML EXPORT TEXT statements.

If the name is a URL, it is used “as is.” Otherwise, the name is examined with an **.xtl** extension and then a **.cob** extension. If the file does not exist using the name as entered, then the RUNPATH environment variable is used to search for the file. If the name component is absent, the current executing COBOL program is searched, followed by calling COBOL programs (if present). Whatever data name (following the "#" character) is present is carried forward.

Example

```
XML RESOLVE MODEL-NAME
          MY-MODEL-DATA-FILE.
IF NOT XML-OK GO TO Z.
```

Directory Management Statements

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

- XML FIND FILE (on page 51)
- XML GET UNIQUEID (on page 52)

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

Directory polling may be used to communicate with message-driven communications systems. It is a technique that may also be used between various RM/COBOL applications.

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

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

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

- When a program is adding XML document files to a directory concurrently with another program that is moving XML document files to different directory using the C library function **rename** or the Windows API function **MoveFile**, it is possible for the wrong file to be moved or for the file to be moved to the wrong location. This failure can occur without the participation of XML Extensions.
- When a large number of XML document files are written to a directory by XML Extensions using XML EXPORT FILE (on page 28), it is possible that files will not be placed in the directory and no error will be returned by the operating system either to XML Extensions or to the program issuing the statement. It appears that the FAT32 file system may be limited to 65,535 files per directory (at least under certain conditions). Furthermore, if long filenames are used, multiple directory entries may be needed for each filename, further reducing the number of files per directory.

For these reasons, Liant recommends that directory polling not be used on Windows running with FAT32 file systems. Windows with the NTFS file system and UNIX file systems do not demonstrate this problem.

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.
Extension	A user-specified extension having the format, *.aaa*. Unless *Extension* is specified, the statement looks only for files in the directory that have an extension of ".xml".

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, unless the *Extension* parameter is specified). If there are one or more XML documents in the specified directory, the name of one of the files will be returned in the *FileName* parameter.

If the statement succeeds (the condition `XML-IsSuccess` is true), the XML document specified by the *FileName* parameter may be processed by using XML IMPORT FILE (on page 34).

Before calling XML FIND FILE again (to process the next file), you must call XML REMOVE FILE (on page 46) to delete the XML document that was just processed. Otherwise, the next call to the XML FIND FILE statement may return the same file.

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

Example

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

XML GET UNIQUEID

This statement has the following parameter:

Parameter	Description
UniqueID	The unique value returned by this statement is a string representation having the same format as 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. On Windows systems, the value is an actual UUID. On UNIX systems, the value is a string having the same format as a UUID, but constructed by an internal algorithm. This algorithm uses various components, including the system ID, the start time of the run unit, the current time, and an internal counter, to generate a unique value.

Description

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

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

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

Example

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

State Management Statements

Calls to the following XML statements control several states or conditions, including:

- **Compatibility between current and previous versions.** The XML COMPATIBILITY MODE (on page 55) statement allows version 12 of XML Extensions to be compatible with existing data and applications.
- **Initialization and termination.** Before issuing a call to any other XML Extensions statement, XML INITIALIZE (on page 55) must be called. (If XML INITIALIZE has not been called, any subsequent calls, for example, XML EXPORT FILE, will fail.) Similarly, XML TERMINATE (on page 56) should be called when the COBOL application is finished using XML Extensions statements. (If XML TERMINATE has not been called prior to program termination, there are no consequences.)
- **Empty array occurrences.** As an optimization, trailing "empty" occurrences of arrays are normally not generated by the statements, XML EXPORT FILE (on page 28) or XML EXPORT TEXT (on page 31).

  An empty occurrence of an array is defined to be one where the numeric items have a zero value and the nonnumeric items have a value equivalent to all spaces. This is the default state and is equivalent to calling XML DISABLE ALL-OCCURRENCES (on page 56). It is possible to force all occurrences to be output by calling XML ENABLE ALL-OCCURRENCES (on page 57).
- **COBOL attributes.** For each element generated by the statements, XML EXPORT FILE (on page 28) or XML EXPORT TEXT (on page 31), there is a series of COBOL attributes that describe that element.

  The default state is not to output these attributes. However, it is sometimes necessary for a following activity (such as an XSLT stylesheet transformation) to have access to these attributes (specifically, length and subscript are often important to a follow-on activity). Using XML DISABLE ATTRIBUTES (on page 57) prevents attributes from being written (this is the default). Using XML ENABLE ATTRIBUTES (on page 58) forces these attributes to be written.
- **Document caching.** XML documents, such as XSLT stylesheets, templates, and schemas, are normally considered to be static during the use of a production version of the application. That is, they are generated when the application is developed and are not modified until the application is modified.

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

  If XSLT stylesheets, templates, or schemas are being generated dynamically, the user may selectively enable or disable caching. Executing XML ENABLE CACHE (on page 59), which sets the default behavior, enables caching of documents. Executing XML DISABLE CACHE (on page 58) disables caching, thus forcing all documents to be loaded each time they are referenced. Executing XML FLUSH CACHE (on page 59) flushes all documents and local memory from the cache without changing the state of caching (that is, if caching was enabled it remains enabled). Executing any of the following statements causes the contents of the cache to be flushed: XML INITIALIZE, XML ENABLE CACHE, XML DISABLE CACHE, XML FLUSH CACHE, and XML

TERMINATE. Executing XML ENABLE CACHE, XML DISABLE CACHE, or XML FLUSH CACHE also causes local memory to be flushed.

For more information, see Memory Management with XML Extensions (on page 26).

- **CodeBridge flags.** The data conversions performed by the statements, XML EXPORT FILE (on page 28), XML EXPORT TEXT (on page 31), XML IMPORT FILE (on page 34), and XML IMPORT TEXT (on page 36), use the CodeBridge library (which is built into the RM/COBOL runtime) to perform these conversions. By default, the following CodeBridge flags are set: PF_TRAILING_SPACES, PF_LEADING_SPACES, PF_LEADING_MINUS, and PF_ROUNDED.

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

  XML GET FLAGS (on page 60) and XML SET FLAGS (on page 64) are available to alter these defaults. Refer to the CodeBridge manual for a more complete presentation of the CodeBridge conversion library.

- **Internal character encoding.** Characters within alphanumeric data elements in a COBOL program are normally encoded using the conventions of underlying operating systems. Under some conditions, it may be desirable to encode these same data items using UTF-8 encoding. (UTF-8 is a format for representing Unicode.) XML SET ENCODING (on page 63) is provided to switch between the local encoding format and UTF-8.

  Note Both the UNIX and Windows implementations of XML Extensions allow the in-memory representation of element content to use UTF-8 encoding. This may be useful for COBOL applications that wish to pass UTF-8-encoded data to other processes. XML documents are normally encoded using Unicode. XML Extensions always generates UTF-8 data. For more information, see COBOL and Character Encoding (on page 70) and XML and Character Encoding (on page 83).

- **Tracing.** Trace information can be generated to a designated file using the XML TRACE statement (on page 60).

- **Stylesheet parameters.** The passing of parameters to stylesheets can be controlled by the statements XML SET XSL-PARAMETERS (on page 64) and XML CLEAR XSL-PARAMETERS (on page 65).

XML COMPATIBILITY MODE

This statement has the following parameter:

Parameter	Description
Flags	0 = Compatibility mode is off. 1 = Compatibility mode is on.

Description

The XML COMPATIBILITY MODE statement allows version 12 XML Extensions to be compatible with existing data and applications by inserting <root> as the top-level entry in a document during an export operation. While versions of XML Extensions prior to version 12 required that <root> be the top-level element of a document, version 12 and later of XML Extensions will tolerate either the presence or absence or the <root> element. The <root> element (compatibility mode on) in version 12 is generally necessary only when external stylesheets refer to the <root> element and the user does not wish to modify the stylesheets to eliminate those references.

Example

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

XML INITIALIZE

This statement has no parameters.

Description

The XML INITIALIZE statement opens a session with XML Extensions. It ensures that the RM/COBOL runtime system is the required version (12 or greater) and retrieves required information from the runtime system. RM/COBOL runtime version 12 or greater is required because information needed by XML Extensions is not available in prior runtime versions. The underlying XML parser is also initialized.

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

A status value is returned in the data item `XML-data-group`, which is defined in the copy file, **lixmldef.cpy**. Errors can occur if the RM/COBOL runtime version is not 12 or greater, or the underlying XML parser initialization fails. It is not considered an error to execute an XML INTIALIZE statement when XML Extensions has already been initialized and not terminated.

Example

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

XML TERMINATE

This statement has no parameters.

Description

The XML TERMINATE statement flushes the document cache and closes a session with XML Extensions. The interface to the underlying XML parser is also closed. Any memory blocks that were allocated by XML Extensions 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 under the following circumstances:

- The calls to free memory fail.
- The underlying XML parser termination fails.

It is not considered an error to execute an XML TERMINATE statement when XML Extensions has not been initialized or has already been terminated.

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 (COBOL table) occurrences not to be generated by the statements, XML EXPORT FILE (on page 28) and XML EXPORT TEXT (on page 31). An empty array is one in which all numeric elements have a zero value and all nonnumeric elements have a value of all spaces.

There is some interoperation with the statements, XML DISABLE ATTRIBUTES (on page 57) and XML ENABLE ATTRIBUTES (on page 58). If attributes are enabled (that is, XML ENABLE ATTRIBUTES has been called), then all empty occurrences are not generated. If attributes are disabled (the default state or if XML DISABLE ATTRIBUTES has been used), then all trailing empty occurrences are not generated. If attributes are enabled, then the subscript is present and so leading, or intermediate, empty occurrences are not needed as placeholders to ensure that the correct subscript is calculated.

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

Example

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

XML ENABLE ALL-OCCURRENCES

This statement has no parameters.

Description

The XML ENABLE ALL-OCCURRENCES statement causes all occurrence of an array (COBOL table) to be generated by the statements, XML EXPORT FILE (on page 28) and XML EXPORT TEXT (on page 31), 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.

See XML DISABLE ALL-OCCURRENCES (on page 56) regarding the behavior of array (COBOL table) output when 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 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

See XML DISABLE ALL-OCCURRENCES (on page 56) regarding the behavior of array (COBOL table) output when attributes are enabled or disabled.

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

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

Example

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

XML DISABLE CACHE

This statement has no parameters.

Description

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

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

Example

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

XML ENABLE CACHE

This statement has no parameters.

Description

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

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

Example

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

XML FLUSH CACHE

This statement has no parameters.

Description

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

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

Example

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

XML GET FLAGS

This 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 GET FLAGS statement retrieves the setting of the flags that are used for internal data conversion. Valid flag values are specified in the copy file, **lixmldef.cpy**. The initial setting of the flags has the following flag values set: PF-Leading-Spaces, PF-Trailing-Spaces, PF-Leading-Minus, and PF-Rounded. The setting of the flags can be changed with the XML SET FLAGS statement.

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

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

Example

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

XML TRACE

This statement has the following parameters:

Parameter	Description
Flags	0 = Turn tracing off and keep any existing trace file. 1 = Turn tracing on and keep any existing trace file. 2 = Turn tracing off and delete any existing trace file. 3 = Turn tracing on and delete any existing trace file.
[File-name]	Optionally specifies the name of the trace file. If omitted, a value of "XMLTrace.log" is assumed.

Description

The XML TRACE statement generates trace information to a designated file. The statement name and parameter values (as well as the calling program name and the time executed) are recorded on entry. Updated parameter values are displayed on exit.

Examples

Without Trace File Parameter:

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

Showing Optional Trace File Parameter:

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

Showing Trace Output for the Execution of an XML IMPORT FILE Statement:

```
XMLImportFile – entry
 DocumentName[test83i1]
 ModelFileDataName[./code/test83.cob#test-83//test-83-n2-n2//cp1]
 StyleSheetName[]
Date-Time: Tue Apr 17 11:56:16 2007
Called from line 1151 in TEST-83-N2-
N2(C:\xmlext\root\rmc85\test\xmlext\code\TEST83.COB), compiled
2007/04/17 11:55:24.
XMLImportFile – exit
 Status[0]

FullDocumentName[C:\xmlext\root\rmc85\test\xmlext\xml\test83i1.xml]
FullModelFileName[C:\xmlext\root\rmc85\test\xmlext\code\TEST83_TEST-83.xtl]
ModelDataName[test-83//test-83-n2-n2//cp1]
FullStyleSheetName[]
```

XML GET STATUS-TEXT

This statement has no named parameters.

Description

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

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

- `XML-IsSuccess`. A successful completion occurred (no informative, warning, or error messages).
- `XML-OK`. An OK (or satisfactory) completion occurred, including informative or warning messages.
- `XML-IsDirectoryEmpty`. An informative status indicating that XML FIND FILE (see page 51) found no XML documents in the indicated directory.

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

Example

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

Note In the **lixmldef.cpy** copy file, the definition of the `XML-StatusText` field may be edited from the default of 80 to change the size of the buffer used to contain XML status information. See Displaying Status Information (on page 77).

XML SET ENCODING

This statement has the following parameter:

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

Description

The XML SET ENCODING statement allows the developer to specify the character encoding of data within a COBOL data structure. The developer may use this statement to switch between the local character encoding and UTF-8. On Windows, the local character encoding matches the native character set of the runtime; that is, it is specified by the Windows ANSI code page or Windows OEM code page depending on the native character set of the runtime system (see the *RM/COBOL User’s Guide* for how to select the native character set for the runtime system). On UNIX, the local character encoding is specified by the value of the RM_ENCODING environment variable, with a default of RM_LATIN_9 if the variable is not defined.

Note If the value of the *Encoding* parameter specifies “utf-8”, the RM_ENCODING environment variable (on page 70) is ignored. For more information on this environment variable, see COBOL and Character Encoding (on page 70).

Although the XML SET ENCODING statement does not affect the character encoding of the XML document, it does affect the character encoding of the data in the COBOL program. For more information, see Data Representation (on page 69).

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

Example

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

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

XML SET FLAGS

This 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 establishes the setting of the flags that are used for internal data conversion. Valid flag values are specified in the copy file, **lixmldef.cpy**. The initial setting of the flags has the following flag values set: PF-Leading-Spaces, PF-Trailing-Spaces, PF-Leading-Minus, and PF-Rounded.

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

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

Example

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

XML SET XSL-PARAMETERS

This statement has the following parameter:

Parameter	Description
Parameter list	A list containing at least one name/value pair to be used whenever transform operations are performed.

Description

The XML SET XSL-PARAMETERS statement passes a list of name/value pairs to XML Extensions, where they are stored until one of the following occurs:

- They are replaced by a subsequent execution of an XML SET XSL-PARAMETERS statement.
- They are cleared by executing an XML CLEAR XSL-PARAMETERS statement.
- They are cleared by flushing the cache (the statements XML INITIALIZE, XML ENABLE CACHE, XML DISABLE CACHE, XML FLUSH CACHE, and XML TERMINATE all clear the cache).
- The COBOL run-unit terminates.

The saved parameters are used whenever any of the following transform operations occur:

- The XML TRANSFORM FILE statement is executed.
- The XML EXPORT FILE, XML EXPORT TEXT, XML IMPORT FILE, or XML IMPORT TEXT statements reference an optional stylesheet.

A maximum of 20 name/value pairs may be specified. If more than 20 pairs are specified or the parameters are not specified as pairs, an error will be reported.

Example

```
XML SET XSL-PARAMETERS
    "MY-COUNT", 7.
IF NOT XML-OK GO TO Z.
```

XML CLEAR XSL-PARAMETERS

This statement has no parameters.

Description

The XML CLEAR XSL-PARAMETERS statement clears all sets of name/value pairs that have been stored in XML Extensions by the XML SET XSL-PARAMETERS statement.

Example

```
XML CLEAR XSL-PARAMETERS.
IF NOT XML-OK GO TO Z.
```

Chapter 4: COBOL Considerations

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

- File management (see the following topic)
- Data conventions (on page 69)
- Copy files (on page 75)
- Limitations (on page 78)
- Optimizations (on page 80)

File Management

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

- Automatic search for files (as discussed below)
- File naming conventions (on page 68)

Automatic Search for Files

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

- The RM/COBOL runtime support for resolving leading or subsequent names in a path name is not provided by XML Extensions when locating files. That is, XML Extensions does not honor the RESOLVE-LEADING-NAME or RESOLVE-SUBSEQUENT-NAMES keywords of the RUN-FILES-ATTR configuration record.
- If the RUNPATH environment variable contains UNC references (directory names beginning with "//" or "\\"), XML Extensions will skip those names. UNC references typically refer to foreign file systems that are accessed through RM/InfoExpress. These names are skipped in order to avoid server performance degradation.

- The RUNPATH environment variable is also searched to locate input XML data document files and all external XSLT stylesheet files.

File Naming Conventions

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

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

External XSLT Stylesheet File Naming Conventions

External XSLT stylesheets may be referenced by XML Extensions. If the filename parameter supplied by the COBOL program does not contain an extension, the value **.xsl** is added to the filename.

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

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

Other Input File Naming Conventions

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

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

Data Conventions

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

- Data representation (as discussed below)
- FILLER data items (on page 71)
- Missing intermediate parent names (on page 72)
- Sparse COBOL records (on page 75)

Data Representation

COBOL numeric data items are represented in XML as numeric strings. 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 text strings and have trailing spaces removed (or leading spaces, if the item is described with the JUSTIFIED phrase). Note, however, that in edited data items (on page 79), leading and trailing spaces are preserved. 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.

Note For more information, see Handling Spaces and Whitespace in XML (on page 86).

On Windows platforms, nonnumeric displayable data are normally encoded using Microsoft’s OEM or ANSI 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 or ANSI data format. If the XML SET ENCODING statement (on page 63) is used to specify “UTF-8”, then the internal data format is UTF-8. For more information, see the discussion of Windows Character Encoding (on page 70).

On UNIX platforms, nonnumeric displayable data are normally encoded using a “local” character encoding that the UNIX system uses. Typically, this may be Latin-1 or Latin-9. On output, these data are converted to the standard Unicode 8-bit transformation format, UTF-8. On input, data is converted to the systems internal format. If the XML SET ENCODING statement is used to specify “UTF-8”, then the internal data format is UTF-8. For more information on selecting an appropriate “local” character encoding, refer to the discussion of UNIX Character Encoding (on page 70).

COBOL and Character Encoding

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

RM_ENCODING Environment Variable

The RM_ENCODING environment variable is used to specify the "local" character encoding. This environment variable is ignored if the XML SET ENCODING statement (on page 63) sets the encoding to UTF-8. The interpretation of this environment variable also varies between Windows and UNIX character encoding, as discussed in the next topics.

Windows Character Encoding

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

Note Microsoft originally introduced OEM character encoding for MS-DOS. While there are multiple OEM code pages in use, the Windows operating system provides interfaces that allow conversion between the OEM code page in use and Unicode. XML Extensions does not need to differentiate between code pages. In version 9 and later of the runtime system, the ANSI code page can be selected as the native character set, in which case, XML Extensions uses the ANSI code page in use for the conversion to/from Unicode when using the local character encoding.

UNIX Character Encoding

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

The RM_ENCODING environment variable (on page 70) may specify the built-in and predefined values of RM_LATIN_1 and RM_LATIN_9. These values are used to designate that either Latin-1 or Latin-9 is being used as the local character encoding. Internal translation functions convert between either Latin-1 or Latin-9 (in COBOL memory) and UTF-8 (in the XML document). The value of the environment variable is case insensitive, with hyphen and underscore characters being optional. For example, "RM_LATIN_9", "Rm-Latin-9", and "rmlatin9" are equivalent.

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

If the value of the RM_ENCODING environment variable is specified with a value that is not RM_LATIN_1 or RM_LATIN_9, then the value that is passed must be a name recognized by

the **iconv** library. The **iconv** library can perform other conversions. In this case, the spelling may need to be exact (for example, the value may be case sensitive, and hyphens and underscores would be required). The exact spelling of the value of the RM_ENCODING environment variable is specific to the **iconv** library on the platform in use.

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

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

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

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

FILLER Data Items

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

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

For example, the following COBOL sequence:

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

generates the following well-formed XML sequence:

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

FILLER data items, however, are treated differently than named data. All leading and/or trailing spaces are preserved, so that the length of the data is the same as the COBOL data length. For more information, see Handling Spaces and Whitespace in XML (on page 86).

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,
). XHTML (eXtensible HyperText Markup Language) is based on HTML 4, but with restrictions such that an XHTML document is also a well-formed XML document. For example, the following COBOL sequence:

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

generates the following well-formed XML sequence:

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

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

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

generates the following syntactically malformed XML sequence:

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

Whenever FILLER data items are present in a data item that is referenced by the XML EXPORT statements, the resulting document is checked to ensure that the resultant XML document is well-formed. When the document is not well-formed, an appropriate status value is returned to the COBOL program.

Missing Intermediate Parent Names

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

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

- Unique element names (on page 73). Use this technique to determine whether the element name is unique.
- Unique identifier (on page 74). Use this method to determine whether the unique identifier (uid) attributes of the element name are provided. If this is true, then the intermediate parent names may also be generated.

Unique Element Names

Consider the following COBOL data structure:

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

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

```
<?xml version="1.0" encoding="UTF-8" ?>
<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>
</liant-address>
```

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" ?>
<liant-address>
  <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>
</liant-address>
```

Unique Identifier

The unique identifier (uid) attribute is generated by an XML EXPORT FILE (on page 28) or XML EXPORT TEXT (on page 31) statement if XML attributes are enabled. Attributes may be enabled by using the XML ENABLE ATTRIBUTES statement (on page 58) before the XML EXPORT statements.

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

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

A well-formed “flattened” version of an XML document that could also be imported into this structure is displayed below. The uid attributes were captured from an XML document (such as the one shown previously) that was generated by an XML EXPORT statement. These attributes may be captured by an XSLT stylesheet or other process, and then added again before the XML IMPORT FILE (on page 34) or XML IMPORT TEXT (on page 36) 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" ?>
<liant-address>
  <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>
</liant-address>
```

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>
```

Copy Files

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

- Statement definitions (as discussed in the following topic)
- REPLACE statement considerations (on page 76)
- Displaying status information (on page 77)
- Application termination (on page 77)

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 at the beginning of the Working-Storage Section of the source program. This copy file copies the remaining copy files used by XML Extensions. In general, do not modify or edit the contents of this copy file or the copy files that it copies (**lixmdef.cpy** and **lixmlrpl.cpy**).

REPLACE Statement Considerations

The copy file, **lixmlall.cpy**, contains a REPLACE statement to define the XML statements. A COBOL REPLACE statement overrides any lexically preceding REPLACE statement. Thus, in cases where the user's program contains a REPLACE statement, it may not be possible to use the **lixmlall.cpy** file. For this reason, the **lixmlrpl.cpy** copy file, which is copied by the **lixmlall.cpy** file, is provided as part of XML Extensions. The **lixmlrpl.cpy** file contains the operands of the REPLACE statement needed to define the XML statements, but not the REPLACE statement itself. Accordingly, the user's REPLACE statement may be augmented by copying **lixmlrpl.cpy** into the REPLACE statement as follows:

```
REPLACE
  *> include user's replacements here
  COPY "lixmlrpl.cpy".  *> define XML statements
  . *> end of combined REPLACE statement

COPY "lixmldef.cpy".  *>  XML data definitions
```

When this is done, the **lixmlall.cpy** file need not be copied in the source program.

Note If there are multiple REPLACE statements in your source program, each REPLACE statement that precedes any XML statements needs to copy the **lixmlrpl.cpy** file into the REPLACE statement to preserve the statements for replacement.

The InstantSQL product has a copy file, **lisqlall.cpy**, which contains a REPLACE statement to define the SQL statements. In cases where InstantSQL is used with XML Extensions, neither the **lixmlall.cpy** nor the **lisqlall.cpy** copy file should be used. Instead, create a copy file (for example, named **isqlxml.cpy**) with the following contents:

```
REPLACE
  *> optionally include user's replacements
  COPY "lisqlrpl.cpy".  *> define SQL statements
  COPY "lixmlrpl.cpy".  *> define XML statements
  . *> end of combined REPLACE statement

COPY "lisqldef.cpy".  *>  SQL data definitions
COPY "lixmldef.cpy".  *>  XML data definitions
```

Use this copy file in place of **lixmlall.cpy** and **lisqlall.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. For example, the definition of the `XML-StatusText` field in the **lixmldef.cpy** copy file may be altered from the default of 80 to change the size of the buffer used to contain XML status information.

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

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

Application Termination

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

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

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

The following logic is sufficient to successfully terminate XML Extensions:

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

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

Anonymous COBOL Data Structures

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

Limitations

This section describes the limitations of XML Extensions 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), as discussed in the following topic
- Edited data items (on page 79)
- Wide and narrow characters (on page 79)
- Data item size (on page 79)
- Data naming (on page 79)
- OCCURS restrictions (on page 80)
- Reading, writing, and the Internet (on page 80)

Data Items (Data Structures)

The XML IMPORT FILE, XML IMPORT TEXT, XML EXPORT FILE, and XML EXPORT TEXT statements operate on a single COBOL data item. This data item is the second command line parameter when using the optional **slicexsy** utility. As you would expect, this

data item may be (and usually will be) a group item. The COBOL program must move all necessary data to the selected data item before using the XML EXPORT FILE (on page 28) or XML EXPORT TEXT (on page 31) statements and retrieve data from the data item after using the XML IMPORT FILE (on page 34) or XML IMPORT TEXT (on page 36) statement.

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

1. REDEFINES and RENAMES clauses are not allowed.
2. FILLER data items must be nonnumeric.
3. The data item must be the same size or larger than the data item specified when building the model files with the **slicexsy** utility, but it is not required to be the same data item. For additional information, see Anonymous COBOL Data Structures (on page 78).

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. For more information, see Handling Spaces and Whitespace in XML (on page 86).

Wide and Narrow Characters

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

Data Item Size

By its nature, XML has no limits on data item size. COBOL does have size limitations for its data items. Many XML documents have been standardized and such standards include limitations on data items, but the COBOL program must still be written to deal with data item size constraints. When a nonnumeric data item is truncated on import, a warning status value is produced by XML Extensions.

Data Naming

While the COBOL language allows a data-name to begin with a digit, XML does not allow an element name to begin with a digit. For example, the following line defines a valid COBOL data-name, but when using XML Extensions, the data-name will result in an invalid XML element name:

```
03 1099-something-field
```

The RM/COBOL compiler will not detect the issue with the data-name with respect to XML Extensions. However, XML Extensions will detect the problem at runtime and report the

error. A workaround that avoids the need to modify any COBOL Procedure Division code when data-names begin with a digit is to add a non-digit initial character to the data-name and then redefine that data item with the original data-name, as in the following:

```
03 x1099-something-field      PIC X(10).
03 1099-something-field       REDEFINES x1099-something-field
   SAME AS x1099-something-field.
```

The data-name `1099-something-field` will result in the Procedure Division compiling successfully and the `x1099-something-field` will result in a valid element name for XML Extensions.

OCCURS Restrictions

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

Reading, Writing, and the Internet

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

Optimizations

Some optimizations have been added to XML Extensions to improve performance and reduce the size of the generated documents. Refer also to Chapter 3: *XML Extensions Statements Reference* (on page 25) for more information.

Occurs Depending

As expected, on output, the XML EXPORT FILE and XML EXPORT TEXT statements will limit the number of occurrences of a group to the value of the DEPENDING variable. Additional occurrences may be omitted if they contain no data. For more information, see Empty Occurrences (on page 81).

On input, the XML IMPORT FILE and XML IMPORT TEXT statements will store the value of the DEPENDING variable. The XML IMPORT FILE and XML IMPORT TEXT statements will also store all occurrences in the document (up to the maximum occurrence limit), regardless of the value of the DEPENDING variable. However, if a schema file is generated by the optional **slicexsy** utility, as described in Appendix D: *slicexsy Utility Reference* (on page 189), the schema file will report an error if not all of the elements specified by the DEPENDING variable are present.

Empty Occurrences

On output, the XML EXPORT FILE (on page 28) or XML EXPORT TEXT (on page 31) statements recognize occurrences within a group that contain only spaces and zeros. Specifically, an empty data item is an alphanumeric item that contains either all spaces or zero characters, or a numeric item 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 only spaces and zeros.

You may enable all occurrences using the XML ENABLE ALL-OCCURRENCES (on page 57) statement, when generating the document (with XML export operations).

Cached XML Documents

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

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

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

Several XML statements may be used to enable or disable document caching. These statements include: XML ENABLE CACHE (on page 59), XML DISABLE CACHE (on page 58), and XML FLUSH CACHE (on page 59). By default, caching is enabled.

Chapter 5: XML Considerations

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

- XML and character encoding (as discussed in the following topic)
- Document type definition support (on page 84)
- XSLT stylesheet files (on page 85)
- Handling spaces and whitespace in XML (on page 86)
- Schema files (on page 87)

XML and Character Encoding

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

The XML SET ENCODING statement allows the developer to specify the character encoding of data within a COBOL data structure. The developer may use this statement to switch between the local character encoding and UTF-8. Note that even though the XML SET ENCODING statement does not affect the character encoding of the XML document, it does affect the character encoding of the data in the COBOL program. For more information, see Data Representation (on page 69).

Document Type Definition Support

It is possible to specify a document type definition (DTD) when exporting XML documents. Sometimes, when the exported document is destined to be used as an HTML Web page, it is desirable to include references to external entities in the FILLER (that is, unescaped) data. A DTD can be used to define these entities that are referred to by the values of FILLER data items in the COBOL data structure being exported. For example, if the COBOL data structure contains Web page text that includes a trademark, one or more FILLER data items may have entity references for the trademark entity (™). The trademark entity is not among the predefined XML entities, which are presumed to be declared in all XML documents. (The predefined XML entities are lt, gt, amp, apos, and quot.) If the COBOL data structure containing the trademark entity reference is exported, it will not be valid XML unless a DTD is provided that causes the trademark entity to be declared. XML entities can be declared only within a DTD, which may be specified only between the XML header and the root element of the document.

Note If the exported data is processed by an XML stylesheet (XSLT) and the stylesheet contains HTML entity references, the stylesheet itself must include a DTD that defines the entities. Furthermore, it is best practice to make sure that the HTML generated by the stylesheet also contains a DTD defining the entities. In general, it is best to avoid using HTML entities in the XML processing chain.

The XML EXPORT FILE (on page 28) and XML EXPORT TEXT (on page 31) statements allow specifying a document prefix parameter. The document prefix parameter provides a string, which is exported between the XML header and the first element of the document. This string may specify a DTD when one is needed. The DTD may declare entities in the internal subset directly in the provided DTD, or it may specify an external subset through a URL reference. For example, the   entity is declared in the HTML markup declarations, which can be accessed with a prefix of the form:

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

In contrast, Example C: Export File with Document Prefix (on page 170), demonstrates using an internal subset to declare entities in the DTD itself.

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

Note In the Windows implementation, the Microsoft MSXML parser 4.0 ignores the DTD when validating an XML document against a schema file. Any entities declared in the DTD will not be defined and cannot be referenced. If any entities other than the predefined XML entities are referenced, the document is not well-formed and will fail to load, much less validate. Any XML document that contains entity references, other than the predefined XML

entities, must be transformed with an XSLT stylesheet prior to validation against a schema file when using the Microsoft MSXML parser 4.0. For more information, see Schema Files (on page 87).

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

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

XSLT Stylesheet Files

XSLT (Extensible Stylesheet Language Transformations) stylesheet files are used to transform an XML document into another XML document or another type of document—not necessarily in XML format; for example, HTML, PDF, RTF, and so forth. An XSLT stylesheet is an XML document. XML Extensions has a specific statement, XML TRANSFORM FILE (on page 39), which is used for performing XSLT stylesheet transformations. In addition, the import and export statements, XML IMPORT FILE (on page 34), XML IMPORT TEXT (on page 36), XML EXPORT FILE (on page 28), and XML EXPORT TEXT (on page 31), allow an external XSLT stylesheet to be specified as a parameter, making it possible to transform a document while importing or exporting XML documents.

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

The recommended course of action is to use an external XSLT stylesheet file to transform between the COBOL-generated XML document format and the expected document format. XSLT stylesheets are extremely powerful.

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

Handling Spaces and Whitespace in XML

XML Extensions normally strips trailing spaces from COBOL data items when exporting data and restores trailing spaces to COBOL data items when importing data. Leading spaces are also removed and added for justified data items. This default behavior can be modified using the XML SET FLAGS statement, but the default behavior is generally best. The normal treatment of leading and trailing spaces does not apply to FILLER data items (on page 71) or edited data items (on page 79).

Once the data is in XML, further consideration must be given to XML treatment of whitespace, which includes spaces, carriage returns, and line feeds. XML provides a built-in attribute named `xml:space`, which takes a value of "preserve" or "default." The value "preserve" specifies that whitespace in an element should be preserved. The value "default" specifies that leading and trailing whitespace may be removed and embedded whitespace may be normalized to a single space wherever it occurs. The value "default" is the default treatment of whitespace in XML and is generally not changed unless one is trying to produce poetry or other special output.

When using XSLT stylesheets, the `xsl:strip-space` and `xsl:preserve-space` elements indicate how whitespace should be handled while transforming a document. Preserving whitespace is the default, but tools that generate XSLT stylesheets might insert `xsl:strip-space elements`.

Be aware that when documents are transformed to HTML for display by a browser, many browsers strip whitespace as they are allowed to do. Displaying data in tables is generally necessary to align data in columns rather than using whitespace as it is generally done in COBOL report output.

Schema Files

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

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

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

The third approach applies when the optional **slicexsy** utility is run with any of the command line schema options other than the default, **-sn** (schema none). The schema file generated by the **slicexsy** utility is used to validate XML documents that are loaded by the XML VALIDATE FILE or XML VALIDATE TEXT statements.

Notes

- When the **slicexsy** utility generates a schema file, it also generates a stylesheet. Both should be deployed with the application.
- In the Windows implementation, the Microsoft MSXML parser 6.0 ignores the document type definition (DTD) when validating an XML document against a schema file. Any entities declared in the DTD will not be defined and cannot be referenced. If any entities other than the predefined XML entities are referenced, the document is not well-formed and will fail to load, much less validate. Thus, when a DTD is generated to define entities in an exported document, the exported document should be transformed prior to being imported so as not to contain entity references. For information on generating a DTD to define entity references, see Document Type Definition Support (on page 84).
- On UNIX systems, the underlying XML parser, **libxml**, does not support schema validation.

Appendix A: XML Extensions Examples

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

Note You will find it instructive to examine these examples first before referring to Appendix B: *XML Extensions Sample Application Programs* (on page 177), which describes how to use and access the more complete application programs that are included with the XML Extensions development system.

The following example programs are provided in this appendix:

- Example 1: Export File and Import File (see page 90)
- Example 2: Export File and Import File with XSLT Stylesheets (see page 95)
- Example 3: Export File and Import File with OCCURS DEPENDING (see page 103)
- Example 4: Export File and Import File with Sparse Arrays (see page 109)
- Example 5: Export Text and Import Text (see page 120)
- Example 6: Export File and Import File with Directory Polling (see page 126)
- Example 7: Export File, Test Well-Formed File, and Validate File (see page 134)
- Example 8: Export Text, Test Well-Formed Text, and Validate Text (see page 140)
- Example 9: Export File, Transform File, and Import File (see page 147)
- Example A: Diagnostic Messages (see page 154)
- Example B: Import File with Missing Intermediate Parent Names (see page 162)
- Example C: Export File with Document Prefix (see page 170)

Additionally, three batch files are provided to facilitate use of the example programs. See Example Batch Files (on page 175).

Example 1: Export File and Import File

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

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 34), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example 1

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 1

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

Line	Statement
1	`rmcobol example1`
2	`start /w runcobol example1 k`

Line 1 compiles the **example1.cbl** source file with an embedded XML-format symbol table.

Line 2 executes **example1.cob**. The K Option suppresses the runtime banner message. On line 2, the `start /w` sequence is included only as good programming practice.

Program Description for Example 1

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

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

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

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

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

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

Data Item for Example 1

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

```
* Title: LIANT.CPY: Liant-Address data structure.
*   XML Extension Version 12.0d.
*
* Copyright (c) 2008 Liant Software Corporation.
*
* You have a royalty-free right to use, modify, reproduce, and
* distribute this COBOL source file (and/or any modified version)
* in any way you find useful, provided that you retain this notice
* and agree that Liant has no warranty, obligations, or liability
* for any such use of the source file.
*
* Version Identification:
*   $Revision: 1441 $
*   $Date: 2005-08-09 10:08:38 -0500 (Tue, 09 Aug 2005) $
*
  01  Liant-Address.
      02  Name          Pic X(64)
             Value "Liant Software Corporation".
      02  Address-1     Pic X(64)
             Value "5914 West Courtyard Drive".
      02  Address-2     Pic X(64) Value "Suite 100".
      02  Address-3.
          03  City      Pic X(32) Value "Austin".
          03  State     Pic X(2) Value "TX".
          03  Zip       Pic 9(5) Value 78730.
      02  Time-Stamp    Pic 9(8).
```

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

Other Definitions for Example 1

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example 1

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

Initialization (Example 1)

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 (Example 1)

COBOL Statement	Description
`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.
`XML EXPORT FILE Liant-Address "liant1" "Liant-Address".`	Execute the XML EXPORT FILE statement specifying: the data item address, the XML document filename, and the *ModelFileName#DataFileName* parameter value.
`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Importing an XML Document (Example 1)

COBOL Statement	Description
`Move Spaces to Liant-Address.`	Ensure that the Liant-Address item contains no data.

`XML IMPORT FILE`
`  Liant-Address`
`  "liant1"`
`  "Liant-Address".`
	Execute the XML IMPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example 1)

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 (Example 1)

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

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

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

Status Display Logic (Example 1)

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

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

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

Execution Results for Example 1

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

COBOL Display (Example 1)

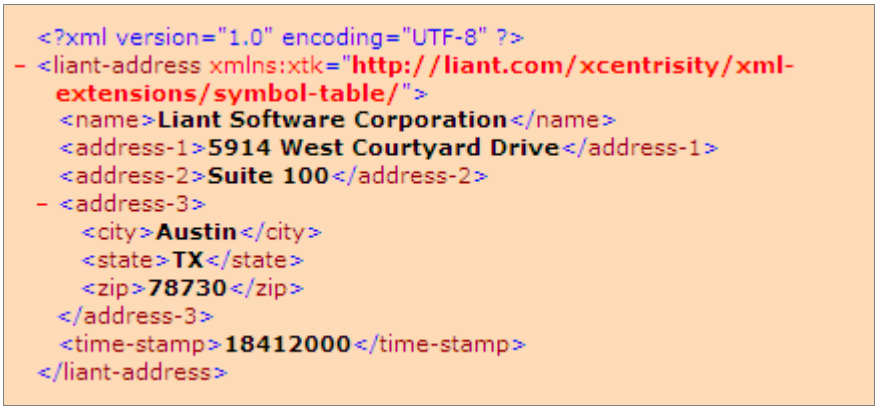
Note Pressing a key will terminate the program.

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

```
Example-1 - Illustrate EXPORT FILE and IMPORT FILE
liant1.xml exported by XML EXPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16423072
liant1.xml imported by XML IMPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16423072

You may inspect 'liant1.xml'

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

XML Document (Example 1)

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



Example 2: Export File and Import File with XSLT Stylesheets

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

This example is almost identical to Example 1: Export File and Import File (on page 90). However, an external XSLT stylesheet is used to transform the exported document into a different format. Similarly, when the document is imported, an external XSLT stylesheet is used to reformat the document into the form that is expected by COBOL. For more information on stylesheets, see XSLT Stylesheet Files (on page 85).

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 34), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

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

Development for Example 2

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 2

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

Line	Statement
1	`rmcobol example2`
2	`start /w runcobol example2 k`

Line 1 compiles the **example2.cbl** source file with an embedded XML-format symbol table.

Line 2 executes **example2.cob**. The K Option suppresses the runtime banner message. On line 2, the `start /w` sequence is included only as good programming practice.

Program Description for Example 2

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

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

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

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

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

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

Data Item for Example 2

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

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "5914 West Courtyard Drive".
    02  Address-2  Pic X(64) Value "Suite 100".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78730.
    02  Time-Stamp Pic 9(8).
```

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

Other Definitions for Example 2

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion.
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion.
```

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

Program Structure for Example 2

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

Initialization (Example 2)

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 (Example 2)

COBOL Statement	Description
`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`  Liant-Address`
`  "liant2"`
`  "Liant-Address"`
`  toext.`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	the *ModelFileName#DataFileName* parameter value,
	and the external XSLT stylesheet name.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Importing an XML Document (Example 2)

COBOL Statement	Description
`Move Spaces to Liant-Address.`	Ensure that the Liant-Address structure contains no data.

`XML IMPORT FILE`
`  Liant-Address`
`  "liant2"`
`  "Liant-Address"`
`  toint.`
	Execute the XML IMPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	the *ModelFileName#DataFileName* parameter value,
	and the external XSLT stylesheet name.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example 2)

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 (Example 2)

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

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

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

Status Display Logic (Example 2)

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

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

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

XSLT Stylesheets for Example 2

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

These external XSLT stylesheets are user-defined and manually generated using a text editor program.

toext.xsl (Example 2)

```
<?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="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 (Example 2)

```
<?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">
    - <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>
    </xsl:template>
  </xsl:stylesheet>
```

Execution Results for Example 2

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

COBOL Display (Example 2)

Note Pressing a key will terminate the program.

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

```
Example-2 - Illustrate EXPORT FILE and IMPORT FILE with XSLT stylesheets
liant2.xml exported by XML EXPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16441288
liant2.xml imported by XML IMPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16441288

You may inspect 'liant2.xml'

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

XML Document (Example 2)

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

```
<?xml version="1.0" encoding="UTF-8" ?>
- <LiantAddress>
    <Information Name="Liant Software Corporation" Address1="5914 West
     Courtyard Drive" Address2="Suite 100" City="Austin" State="TX"
     Zip="78730" />
    <TimeStamp Value="18412225" />
  </LiantAddress>
```

This XML document differs from the document generated in Example 1: Export File and Import File (on page 90). Items that were shown as individual data elements in Example 1 are now shown as attributes of higher-level elements. Notice that this document contains no text. All of the information is contained in the markup.

Example 3: Export File and Import File with OCCURS DEPENDING

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

This program is very similar to Example 1: Export File and Import File (on page 90). However, the data item has been modified so that an OCCURS DEPENDING clause is present.

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 34), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example 3

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 3

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

Line	Statement
1	`rmcobol example3`
2	`start /w runcobol example3 k`

Line 1 compiles the **example3.cbl** source file with an embedded XML-format symbol table.

Line 2 executes **example3.cob**. The K Option suppresses the runtime banner message. On line 2, the `start /w` sequence is included only as good programming practice.

Program Description for Example 3

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

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

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

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

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

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

Data Item for Example 3

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

```
* Title: LIANT3.CPY: Liant-Address data structure.
*   XML Extensions Version 12.0d.
*
* Copyright (c) 2008 Liant Software Corporation.
*
* You have a royalty-free right to use, modify, reproduce, and
* distribute this COBOL source file (and/or any modified version)
* in any way you find useful, provided that you retain this notice
* and agree that Liant has no warranty, obligations, or liability
* for any such use of the source file.
*
* Version Identification:
*   $Revision: 1441 $
*   $Date: 2005-08-09 10:08:38 -0500 (Tue, 09 Aug 2005) $
*
 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 78730.
     02  Address-Lines Pic 9.
     02  Address-Line  Pic X(64)
             Occurs 1 to 5 times
             Depending on Address-Lines.
```

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

A new data item named `Address-Lines` has been added just prior to the Address-Line array. Address-Lines is the depending variable for the array Address-Line.

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

Other Definitions for Example 3

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example 3

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

Initialization (Example 3)

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 (Example 3)

COBOL Statement	Description
`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.

`Move 2 to Address Lines.`
`Move "5914 West Courtyard`
`Drive" to Address-Line(1).`
`Move "Suite 100" to Address-`
`Line (2).`
	Ensure that Address Lines contain proper information.

`XML EXPORT FILE`
`Liant-Address`
`"liant3"`
`"Liant-Address".`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Importing an XML Document (Example 3)

COBOL Statement	Description
`Move Spaces to Liant-Address.`	Ensure that the Liant-Address structure contains no data.

`XML IMPORT FILE`
`Liant-Address`
`"liant3"`
`"Liant-Address".`
	Execute the XML IMPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example 3)

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 (Example 3)

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

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

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

Status Display Logic (Example 3)

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

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

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

Execution Results for Example 3

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

COBOL Display (Example 3)

Note Pressing a key will terminate the program.

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

```
Example-3 - Illustrate EXPORT FILE and IMPORT FILE with OCCURS DEPENDING
liant3.xml exported by XML EXPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16452108
liant3.xml imported by XML IMPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16452108

You may inspect 'liant3.xml'

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

XML Document (Example 3)

Microsoft Internet Explorer may be used to view the generated XML document, **liant3.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
- <liant-address xmlns:xtk="http://liant.com/xcentrisity/xml-
    extensions/symbol-table/">
    <time-stamp>18412383</time-stamp>
    <name>Liant Software Corporation</name>
    <city>Austin</city>
    <state>TX</state>
    <zip>78730</zip>
    <address-lines>2</address-lines>
    <address-line>5914 West Courtyard Drive</address-line>
    <address-line>Suite 100</address-line>
  </liant-address>
```

Example 4: Export File and Import File with Sparse Arrays

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

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

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 34), which reads an XML document (from a file) into a COBOL data item.
- XML ENABLE ATTRIBUTES (on page 58), which causes exported XML document to contain descriptive (COBOL-oriented) attributes.

  Note Although the default is not to add descriptive attributes to an XML document (see XML DISABLE ATTRIBUTES in the next item), among the attributes that may be added is the "subscript" attribute. This attribute contains the one-relative index of the occurrence within the array. When an XML document is imported, this subscript attribute is used (if present) to place the occurrence correctly within the array. If the subscript attribute is not present, then occurrences are assumed to occur sequentially.
- XML DISABLE ATTRIBUTES (on page 57), which causes exported XML documents not to contain descriptive attributes.

  Note The default is not to add descriptive attributes to an XML document.
- XML ENABLE ALL-OCCURRENCES (on page 57), which causes all occurrences of a data item to be exported to an XML document.
- XML DISABLE ALL-OCCURRENCES (on page 56), which causes only certain occurrences to be exported to the XML document.

  Note The default is to export only certain occurrences to the XML document.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example 4

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 4

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

Line	Statement
1	`rmcobol example4`
2	`start /w runcobol example4 k`

Line 1 compiles the **example4.cbl** source file with an embedded XML-format symbol table.

Line 2 executes **example4.cob**. The K Option suppresses the runtime banner message. On line 2, the `start /w` sequence is included only as good programming practice.

Program Description for Example 4

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

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

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

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

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

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

Data Item for Example 4

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

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

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

Other Definitions for Example 4

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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 for Example 4

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

Initialization (Example 4)

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 (Example 4)

COBOL Statement	Description

`XML ENABLE ATTRIBUTES`
`If Not XML-OK Go To Z.`
`XML ENABLE All-OCCURRENCES`
`If Not XML-OK Go To Z.`
	Selectively ENABLE or DISABLE ATTRIBUTES and ALL-OCCURRENCES.

`Initialize Data-Table.`
`Move "B" to X (2).`
`Move  2  to N (2).`
`Move "D" to X (4).`
`Move  4  to N (4).`
	Initialize the Data-Table structure to the preferred values.

`XML EXPORT FILE`
`Data-Table`
`"table1"`
`"Data-Table".`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,
	the XML document filename (table1 – table4),
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Importing an XML Document (Example 4)

COBOL Statement	Description
`Initialize Data-Table.`	Ensure that the data item contains no data.

`XML IMPORT FILE`
`Data-Table`
`"table1"`
`"Data-Table".`
	Execute the XML IMPORT FILE statement specifying:
	the data item address,
	the XML document filename (table1 – table6),
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example 4)

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 (Example 4)

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

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

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

Status Display Logic (Example 4)

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

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

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

Execution Results for Example 4

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

COBOL Display (Example 4)

Note Pressing a key will terminate the program.

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

```
Example-4 - Illustrate EXPORT FILE and IMPORT FILE with sparse arrays
table1.xml exported by XML EXPORT FILE: [ 0B2 0D4 0 0]
table2.xml exported by XML EXPORT FILE: [ 0B2 0D4 0 0]
table3.xml exported by XML EXPORT FILE: [ 0B2 0D4 0 0]
table4.xml exported by XML EXPORT FILE: [ 0B2 0D4 0 0]
table1.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
table2.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
table3.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
table4.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
table5.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]
table6.xml imported by XML IMPORT FILE: [ 0B2 0D4 0 0]

You may inspect 'table1.xml' - 'table6.xml'

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

XML Documents (Example 4)

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

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

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

table1.xml (Example 4)

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

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

```
<?xml version="1.0" encoding="UTF-8" ?>
- <data-table xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-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>
```

table2.xml (Example 4)

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

```
<?xml version="1.0" encoding="UTF-8" ?>
- <data-table xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-table/" compiledTimeStamp="2008-08-08T18:41:24"
  cobtoxmlRevision="1.0" type="xsd:string" kind="GRP" length="14" offset="4"
  uid="Q1_example-4">
  [
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="1" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="1"
      uid="Q4_example-4" />
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="1" uid="Q5_example-4">0</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="2" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="2"
      uid="Q4_example-4">B</x>
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="2" uid="Q5_example-4">2</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="3" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="3"
      uid="Q4_example-4" />
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="3" uid="Q5_example-4">0</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="4" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="4"
      uid="Q4_example-4">D</x>
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="4" uid="Q5_example-4">4</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="5" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="5"
      uid="Q4_example-4" />
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="5" uid="Q5_example-4">0</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="6" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="6"
      uid="Q4_example-4" />
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="6" uid="Q5_example-4">0</n>
  </table-1>
  ]
</data-table>
```

table3.xml (Example 4)

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

```
<?xml version="1.0" encoding="UTF-8" ?>
- <data-table xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-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>
```

table4.xml (Example 4)

The XML ENABLE ATTRIBUTES and XML ENABLE ALL-OCCURRENCES statements are used to determine the content of this file. These statements produce the most verbose listing of occurrences possible. Every occurrence is listed with its attributes.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <data-table xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-table/" compiledTimeStamp="2008-08-08T18:41:24"
  cobtoxmlRevision="1.0" type="xsd:string" kind="GRP" length="14" offset="4"
  uid="Q1_example-4">
  [
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="1" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="1"
      uid="Q4_example-4" />
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="1" uid="Q5_example-4">0</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="2" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="2"
      uid="Q4_example-4">B</x>
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="2" uid="Q5_example-4">2</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="3" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="3"
      uid="Q4_example-4" />
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="3" uid="Q5_example-4">0</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="4" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="4"
      uid="Q4_example-4">D</x>
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="4" uid="Q5_example-4">4</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="5" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="5"
      uid="Q4_example-4" />
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="5" uid="Q5_example-4">0</n>
  </table-1>
  - <table-1 type="xsd:string" kind="GRP" length="2" offset="5" minOccurs="6"
    maxOccurs="6" span="2" subscript="6" uid="Q3_example-4">
    <x type="xsd:string" kind="ANS" length="1" offset="5" subscript="6"
      uid="Q4_example-4" />
    <n type="xsd:decimal" kind="NSU" length="1" offset="6" scale="0"
      precision="1" subscript="6" uid="Q5_example-4">0</n>
  </table-1>
  ]
</data-table>
```

table5.xml (Example 4)

This file was manually generated using a text editor program in order to contain the minimum amount of information possible. Of all the attributes, only the subscript attribute is included. This allows all empty occurrences to be suppressed. In practice, an XSLT stylesheet or other software could generate this kind of document.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <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>
```

table6.xml (Example 4)

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" ?>
- <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>
```

Example 5: Export Text and Import Text

This program first writes (or exports) an XML document as a text string from the content of a COBOL data item. Then the program reads (or imports) the same XML document and places the content in the same COBOL data item. Finally, the text string representation of the XML document is copied to a disk file and the memory block that it occupied is released.

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT TEXT (on page 31), which constructs an XML document (as a text string) from the content of a COBOL data item.
- XML IMPORT TEXT (on page 36), which reads an XML document (from a text string) into a COBOL data item.
- XML PUT TEXT (on page 46), which copies an XML document from a text string to a data file.
- XML FREE TEXT (on page 44), which releases the memory that was allocated by XML EXPORT TEXT to hold the XML document as a text string.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example 5

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 5

The following DOS commands may be entered into a batch file. These commands build and execute **example5.cob**.

Line	Statement
1	`rmcobol example5`
2	`start /w runcobol example5 k`

Line 1 compiles the **example5.cbl** source file with an embedded XML-format symbol table.

Line 2 executes **example5.cob**. The K Option suppresses the runtime banner message. On line 2, the `start /w` sequence is included only as good programming practice.

Program Description for Example 5

This COBOL program illustrates how an XML document is generated from a COBOL data item, and then how the content of an XML document may be converted into COBOL data format and stored in a COBOL data item. This program is similar to Example 1: Export File and Import File (on page 90), except that the XML document is stored as a text string instead of a disk file.

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

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document as defined by the variable, *`Document-Pointer`*, using the XML EXPORT TEXT statement.

Next, the content of the XML document is imported from the file, **liant5.xml**, and placed in the same data item using the XML IMPORT TEXT statement.

Then, the contents of the text string are written to a disk file using the XML PUT TEXT statement. The memory block is deallocated using the XML FREE TEXT statement. The primary aim of using the XML PUT TEXT statement is to make the content of the XML document available as an external file for viewing.

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

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

Data Item for Example 5

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

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "5914 West Courtyard Drive".
    02  Address-2  Pic X(64) Value "Suite 100".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78730.
    02  Time-Stamp Pic 9(8).
```

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

Other Definitions for Example 5

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example 5

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example5.cbl**.

Initialization (Example 5)

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 (Example 5)

COBOL Statement	Description
`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.

`XML EXPORT TEXT`
`  Liant-Address`
`  Document-Pointer`
`  "Liant-Address".`
	Execute the XML EXPORT TEXT statement specifying:
	the data item address,
	the XML document text pointer,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Importing an XML Document (Example 5)

COBOL Statement	Description
`Move Spaces to Liant-Address.`	Ensure that the Liant-Address structure contains no data.

`XML IMPORT TEXT`
`  Liant-Address`
`  Document-Pointer`
`  "Liant-Address".`
	Execute the XML IMPORT TEXT statement specifying:
	the data item address,
	the XML document text pointer,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Copying an XML Document to a File (Example 5)

COBOL Statement	Description

`XML PUT TEXT`
`  Document-Pointer`
`  "liant5".`
	Execute the XML PUT TEXT statement specifying:
	the XML document text pointer
	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 (Example 5)

COBOL Statement	Description
`XML FREE TEXT` `  Document-Pointer.`	Execute the XML FREE TEXT statement specifying the XML document text pointer.
`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example 5)

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 (Example 5)

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

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

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

Status Display Logic (Example 5)

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

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

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

Execution Results for Example 5

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

COBOL Display (Example 5)

Note Pressing a key will terminate the program.

Running the program (**runcobol example5**) produces the following display:

```
Example-5 - Illustrate EXPORT TEXT and IMPORT TEXT
Document exported by XML EXPORT TEXT
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16475280
Document imported by XML IMPORT TEXT
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16475280
Document memory written by XML PUT TEXT
Document memory released by XML FREE TEXT

You may inspect 'liant5.xml'

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

XML Document (Example 5)

Microsoft Internet Explorer may be used to view the generated XML document, **liant5.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
- <liant-address xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-table/">
    <name>Liant Software Corporation</name>
    <address-1>5914 West Courtyard Drive</address-1>
    <address-2>Suite 100</address-2>
  - <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78730</zip>
    </address-3>
    <time-stamp>18412766</time-stamp>
  </liant-address>
```

Example 6: Export File and Import File with Directory Polling

This COBOL program illustrates how a series of XML documents may be placed in a specific directory and how directory polling may be used to process XML documents as they arrive in that specified directory. For more information on directory-polling schemes, see Directory Management Statements (on page 50).

The program first writes (or exports) five XML document files from the content of a COBOL data item. Each document has a unique name and is written to the same directory. Then the program polls the directory looking for an XML document. When one is found, the program reads (or imports) each XML document and places the content in the COBOL data item.

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 34), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.
- XML GET UNIQUEID (on page 52), which is used to generate a unique identifier that can be used to form a filename.
- XML FIND FILE (on page 51), which finds a XML document file in the specified directory (if one is available).
- XML REMOVE FILE (on page 46), which deletes a file.

Development for Example 6

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 6

The following DOS commands may be entered into a batch file. These commands build and execute **example6.cob**.

Line	Statement
1	`rmcobol example6`
2	`start /w runcobol example6 a='\' k`

Line 1 compiles the **example6.cbl** source file with an embedded XML-format symbol table.

Line 2 executes **example6.cob**. The a (argument) option, followed by the directory separator character ('\' on Windows) or ('/' on UNIX), passes a parameter to the RM/COBOL runtime. The K option suppresses the runtime banner message. On line 2, the `start /w` sequence is included only as good programming practice.

Program Description for Example 6

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

The current time, which will become the content of an XML document, is recorded in a COBOL data item. Note that for this example, an elementary data item is used instead of a data item.

Because the name of each file within a directory must be unique, a unique filename is generated using the XML GET UNIQUEID statement. The returned value is combined with other text strings to form a path name using the STRING statement. The current time is placed in the `Time-Stamp` data item using the ACCEPT FROM TIME statement. The XML EXPORT FILE statement is used to output the data item as an XML document. This sequence is repeated until five XML documents have been placed in the specified directory.

Next, the program goes into a loop polling the specified directory. The XML FIND FILE statement is used. If the return status is `XML-IsSuccess`, then a file has been found and the program proceeds to process the file. If the return status is `XML-IsDirectoryEmpty`, then the directory is empty and the program issues a slight delay and then re-issues the XML FIND FILE statement. Any other status indicates an error.

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

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

Data Item for Example 6

The content of the COBOL data item defined in the example, which in this case, is a single data item, is as follows:

```
01  Time-Stamp                  Pic 9(8).
```

This data item stores a time stamp acquired by using the ACCEPT FROM TIME statement.

Other Definitions for Example 6

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example 6

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example6.cbl**.

Initialization (Example 6)

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 (Example 6)

COBOL Statement	Description

`XML GET UNIQUEID`
`  Unique-Name`
`If Not XML-OK Go To Z.`
	Generate a unique identifier.
	If the statement terminates unsuccessfully, go to the termination logic.

`Move Spaces to Unique-File-Name`
`String "Stamp\A" delimited by size`
`  Unique-Name delimited by SPACE`
`  ".xml" delimited by size`
`  into Unique-File-Name.`
	Convert the unique identifier into a path name.

`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`  Liant-Address`
`  "liant6"`
`  "Liant-Address".`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Importing XML Documents by Directory Polling (Example 6)

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` `  = Curr-Time + 100`	Compute Stop-Time to be 1 second after current time.
`  Perform Until 0 > 1`	Inner perform loop. Iterate until Exit Perform.

`  XML FIND FILE`
`  "Stamp"`
`  Unique-File-Name`
	Execute XML FIND FILE parameters:
	directory name
	and filename.

`  If XML-IsSuccess`
`  Exit Perform`
`  End-If`
	If the statement returned success,
	exit the paragraph.

`  If XML-IsDirectoryEmpty`
`  Perform Compute-Curr-Time`
`  If Curr-Time > Stop-Time`
`  Exit Perform`
`  End-If`
`  Call "C$DELAY" Using 0.1`
`  End-If`
	If the statement returns directory empty,
	compute new current time, and
	if the current time is greater than the stop time,
	exit the perform.
	
	Otherwise, do a short time delay.

`  If Not XML-OK`
`  Go To Z`
`  End-If`
	If the statement terminates unsuccessfully,
	go to the termination logic.
	The end of the inner perform loop.

`  End-Perform`	

`  If Curr-Time > Stop-Time`
`  Exit Perform`
`  End-If`
	Check to see if the outer perform loop should terminate.

`  XML IMPORT FILE`
`  Time-Stamp`
`  Unique-File-Name`
`  "Liant-Address"`
	Import the file that was found using:
	the data item,
	the filename,
	and the *ModelFileName#DataFileName* parameter value.

`  If Not XML-OK Go To Z` `  End-If`	If the statement terminates unsuccessfully, go to the termination logic.
`  XML REMOVE FILE` `  Unique-File-Name`	Remove the file that has just been processed; otherwise, find it again.
`  If Not XML-OK Go To Z` `  End-If`	If the statement terminates unsuccessfully, go to the termination logic.
`End-Perform`	The end of the outer perform loop.

Program Exit Logic (Example 6)

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 (Example 6)

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

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

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

Status Display Logic (Example 6)

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

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

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

Execution Results for Example 6

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

COBOL Display (Example 6)

Running the program (**runcobol example6**) produces two displays. The first display occurs after exporting five documents to the **Stamp** directory. The second display takes place after polling the **Stamp** directory and importing the five documents.

First Display

Note Pressing a key will cause the program to continue.

```
Example-6 - Illustrate EXPORT FILE and IMPORT FILE with directory polling
stamp\a{a258c50d-a15e-493b-a29d-cc0e782b5f54}.xml exported by XMLExport
Contents: 10233043
stamp\a{9318803d-1b46-486c-a59b-f7dcc92c4d2f}.xml exported by XMLExport
Contents: 10233054
stamp\a{3a2b7d60-6065-4785-bdf3-ab388992079d}.xml exported by XMLExport
Contents: 10233062
stamp\a{4a4e8482-d3a4-492e-a79d-ec6d967cd4e6}.xml exported by XMLExport
Contents: 10233068
stamp\a{30cd08ac-0edf-4885-a106-4acdcb8caada}.xml exported by XMLExport
Contents: 10233075

You may display the 'stamp' directory

Press a key to continue:
```

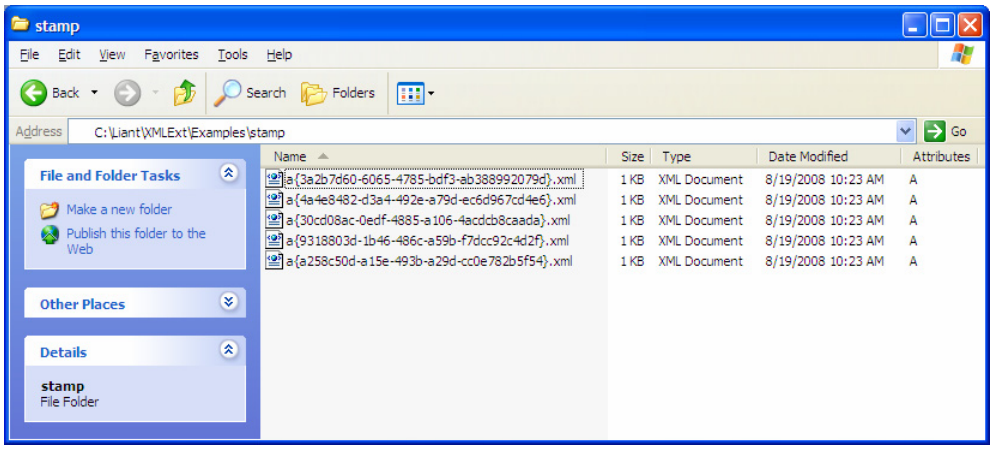
Second Display

Note Pressing a key will terminate the program.

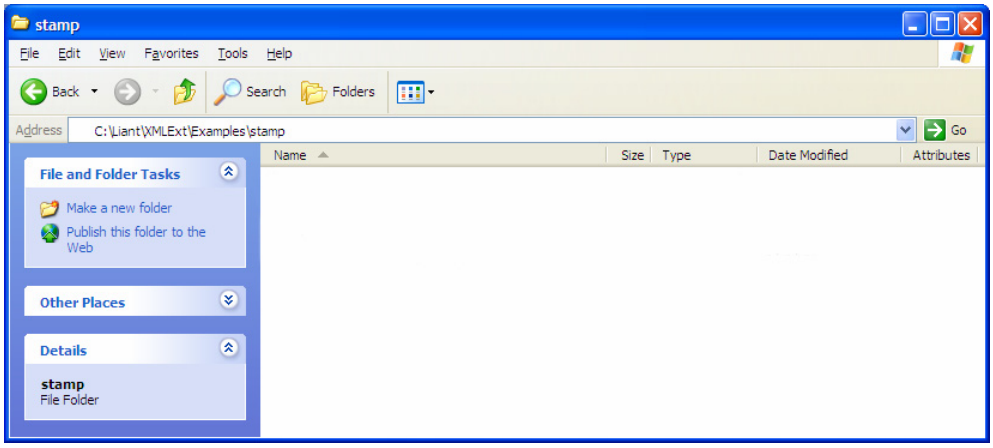
```
 stamp\a{30cd08ac-0edf-4885-a106-4acdcb8caada}.xml imported by XMLImport
Contents: 10233075
 stamp\a{3a2b7d60-6065-4785-bdf3-ab388992079d}.xml imported by XMLImport
Contents: 10233062
 stamp\a{4a4e8482-d3a4-492e-a79d-ec6d967cd4e6}.xml imported by XMLImport
Contents: 10233068
 stamp\a{9318803d-1b46-486c-a59b-f7dcc92c4d2f}.xml imported by XMLImport
Contents: 10233054
 stamp\a{a258c50d-a15e-493b-a29d-cc0e782b5f54}.xml imported by XMLImport
Contents: 10233043

You may now verify that the 'stamp' directory has been emptied

Status: +0001
Informative:  1[0] - indicated directory contains no documents
Called from line 612 in EXAMPLE6(C:\Liant\XMLExt\Examples\EXAMPLE6.COB),
compil\
ed 2008/08/19 10:23:30.
Press a key to terminate.
```

XML Document (Example 6)

Windows Explorer may be used to view the **stamp** directory that contains the five generated XML documents. You can click on any document to see its content.



After continuing the program, the **stamp** directory should empty out as shown.



Example 7: Export File, Test Well-Formed File, and Validate File

This COBOL program illustrates how an XML document is generated from a COBOL data item and then how the syntax and content of an XML document may be verified.

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

Note On UNIX systems, the underlying XML parser, libxml, does not support schema validation.

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML TEST WELLFORMED-FILE (on page 38), which verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE FILE (on page 40), which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example 7

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 7

The following DOS commands may be entered into a batch file. These commands build and execute **example7.cob**.

Line	Statement
1	`rmcobol example7`
2	`slicexsy example7 Liant-Address -ss -bn`
3	`start /w runcobol example7 k`

Line 1 compiles the **example7.cbl** source file with an embedded XML-format symbol table.

Line 2 builds the XML model files from the symbol table information in the RM/COBOL object program. 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** and **example7.xsd**). The option -ss produces a schema file, and the option -bn suppresses the banner message.

Line 3 executes **example7.cob**. The K Option suppresses the runtime banner message. On line 3, the `start /w` sequence is included only as good programming practice.

Note The **example7.xtl** file produced by **slicexsy** is not needed and could be deleted. The slicexsy utility is run for this example only to produce a schema and internal stylesheet to support the XML VALIDATE FILE statement in the example.

Program Description for Example 7

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

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **liant7.xml** using the XML EXPORT FILE statement.

Next, the syntax of **liant7.xml** is verified using the XML TEST WELLFORMED-FILE statement.

Following this, the content of **liant7.xml** is verified using the XML VALIDATE FILE statement.

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

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

For the purposes of this example, both the XML TEST WELLFORMED-FILE and XML VALIDATE FILE statements were used. However, the XML VALIDATE FILE statement also tests an XML document for well-formed syntax.

Data Item for Example 7

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

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "5914 West Courtyard Drive".
    02  Address-2  Pic X(64) Value "Suite 100".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78730.
    02  Time-Stamp Pic 9(8).
```

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

Other Definitions for Example 7

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example 7

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example7.cbl**.

Initialization (Example 7)

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 (Example 7)

COBOL Statement	Description
`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`  Liant-Address`
`  "liant7"`
`  "Liant-Address".`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Verifying Syntax (Example 7)

COBOL Statement	Description
`XML TEST WELLFORMED-FILE` `  "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 (Example 7)

COBOL Statement	Description

`XML VALIDATE FILE`
`  "liant7"`
`  "example7".`
	Execute the XML VALIDATE FILE statement specifying:
	the XML document filename
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example 7)

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 (Example 7)

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

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

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

Status Display Logic (Example 7)

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

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

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

Execution Results for Example 7

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

COBOL Display (Example 7)

Note Pressing a key will terminate the program.

Running the program (**runcobol example7**) produces the following display:

```
Example-7 - Illustrate TEST WELLFORMED-FILE & VALIDATE FILE
liant7.xml exported by XML EXPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16551294
liant7.xml checked by XML TEST WELLFORMED-FILE
liant7.xml validated by XML VALIDATE FILE

You may inspect 'liant7.xml'

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

XML Document (Example 7)

Microsoft Internet Explorer may be used to view the generated XML document **liant7.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
- <liant-address xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-table/">
    <name>Liant Software Corporation</name>
    <address-1>5914 West Courtyard Drive</address-1>
    <address-2>Suite 100</address-2>
  - <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78730</zip>
    </address-3>
    <time-stamp>18413310</time-stamp>
  </liant-address>
```

Example 8: Export Text, Test Well-Formed Text, and Validate Text

This COBOL program illustrates how an XML document is generated from a COBOL data item and then how the syntax and content of an XML document may be verified. Next, the program verifies that the generated document is well-formed. Finally, the program verifies that the content of the document conforms to the schema file that was generated by the **slicexsy** utility.

Note On UNIX systems, the underlying XML parser, libxml, does not support schema validation.

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT TEXT (on page 31), which constructs an XML document (as a text string) from the content of a COBOL data item.
- XML TEST WELLFORMED-TEXT (on page 38), which verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE TEXT (on page 41), which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML PUT TEXT (on page 46), which copies an XML document from a text string to a data file.
- XML FREE TEXT (on page 44), which releases the memory that was allocated by XML EXPORT TEXT to hold the XML document as a text string.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example 8

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 8

The following DOS commands may be entered into a batch file. These commands build and execute **example8.cob**.

Line	Statement
1	`rmcobol example8`
2	`slicexsy example8 Liant-Address -ss -bn`
3	`start /w runcobol example8 k`

Line 1 compiles the **example8.cbl** source file with an embedded XML-format symbol table.

Line 2 builds the XML model files from the symbol table information in the RM/COBOL object program. 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** and **example8.xsd**). The option -ss produces a schema file, and the option -bn suppresses the banner message.

Line 3 executes **example8.cob**. The K Option suppresses the runtime banner message. On line 3, the `start /w` sequence is included only as good programming practice.

Note The **example8.xtl** file produced by **slicexsy** is not needed and could be deleted. The slicexsy utility is run for this example only to produce a schema and internal stylesheet to support the XML VALIDATE TEXT statement in the example.

Program Description for Example 8

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

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document as defined by the variable, *`Document-Pointer`*, using the XML EXPORT TEXT statement.

Next, the syntax of the generated XML document is verified using the XML TEST WELLFORMED-TEXT statement.

Following this, the content of the generated XML document is verified using the XML VALIDATE TEXT statement.

Next, the contents of the text string are written to a disk file using the XML PUT TEXT statement. The memory block is deallocated using the XML FREE TEXT statement. The primary aim of using the XML PUT TEXT statement is to make the content of the XML document available as an external file for viewing.

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

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

For the purposes of this example, both the XML TEST WELLFORMED-TEXT and XML VALIDATE TEXT statements were used. However, the XML VALIDATE TEXT statement also tests an XML document for well-formed syntax.

Data Item for Example 8

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

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "5914 West Courtyard Drive".
    02  Address-2  Pic X(64) Value "Suite 100".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78730.
    02  Time-Stamp Pic 9(8).
```

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

Other Definitions for Example 8

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example 8

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example8.cbl**.

Initialization (Example 8)

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 (Example 8)

COBOL Statement	Description
`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.

`XML EXPORT TEXT`
`  Liant-Address`
`  Document-Pointer`
`  "Liant-Address".`
	Execute the XML EXPORT TEXT statement specifying:
	the data item address,
	the XML document text pointer,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Verifying Syntax (Example 8)

COBOL Statement	Description
`XML TEST WELLFORMED-TEXT` `  Document-Pointer.`	Execute the XML TEST WELLFORMED-TEXT statement specifying the XML document text pointer.
`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Verifying Content (Example 8)

COBOL Statement	Description

`XML VALIDATE TEXT`
`  Document-Pointer`
`  "example8".`
	Execute the XML VALIDATE TEXT statement specifying:
	the XML document text pointer
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Copying an XML Document to a File (Example 8)

COBOL Statement	Description

`XML PUT TEXT`
`  Document-Pointer`
`  "liant8".`
	Execute the XML PUT TEXT statement specifying:
	the XML document text pointer
	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 (Example 8)

COBOL Statement	Description
`XML FREE TEXT` `  Document-Pointer.`	Execute the XML FREE TEXT statement specifying the XML document text pointer.
`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example 8)

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 (Example 8)

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

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

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

Status Display Logic (Example 8)

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

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

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

Execution Results for Example 8

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

COBOL Display (Example 8)

Note Pressing a key will terminate the program.

Running the program (**runcobol example8**) produces the following display:

```
Example-8 - Illustrate TEST-WELLFORMED TEXT and VALIDATE TEXT
Document exported by XML EXPORT TEXT
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16555656
Document checked by XML TEST WELLFORMED-TEXT
Document validated by XML VALIDATE TEXT
Document memory written by XML PUT TEXT
Document memory released by XML FREE TEXT

You may inspect 'liant8.xml'

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

XML Document (Example 8)

Microsoft Internet Explorer may be used to view the generated XML document, **liant8.xml**. The content of this document should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

```
<?xml version="1.0" encoding="UTF-8" ?>
- <liant-address xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-table/">
    <name>Liant Software Corporation</name>
    <address-1>5914 West Courtyard Drive</address-1>
    <address-2>Suite 100</address-2>
  - <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78730</zip>
    </address-3>
    <time-stamp>18413505</time-stamp>
  </liant-address>
```

Example 9: Export File, Transform File, and Import File

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

The program first writes (or exports) an XML document file from the content of a COBOL data item. Next, the document is transformed into another format (the same format as described in Example 2: Export File and Import File with XSLT Stylesheets (on page 95) and then transformed back into the original output format. Then the program reads (or imports) the same XML document and places the content in the same COBOL data item. One additional transform is applied to add in the COBOL attributes to the input document.

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE (on page 34), which reads an XML document (from a file) into a COBOL data item.
- XML TRANSFORM FILE (on page 39), which uses an XSLT stylesheet to modify (transform) an XML document into another format.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example 9

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example 9

The following DOS commands may be entered into a batch file. These commands build and execute **example9.cob**.

Line	Statement
1	`rmcobol example9`
2	`slicexsy example9 Liant-Address -ss -bn`
3	`start /w runcobol example9 k`

Line 1 compiles the **example9.cbl** source file with an embedded XML-format symbol table.

Line 2 builds the XML model files from the symbol table information in the RM/COBOL object program. 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** and **example9.xsd**). The option -ss produces a schema file, and the option -bn suppresses the banner message.

Line 3 executes **example9.cob**. The K Option suppresses the runtime banner message. On line 3, the `start /w` sequence is included only as good programming practice.

Note The **example9.xtl** file produced by **slicexsy** is not needed and could be deleted. The slicexsy utility is run for this example only to produce an internal stylesheet to support one of the XML TRANSFORM FILE statements in the example.

Program Description for Example 9

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

Data is exported from the data item `Liant-Address` (as defined in the copy file, **liant.cpy**) to an XML document with the filename of **liant9a.xml** using the XML EXPORT FILE statement.

Next, the content of the XML document is transformed from the format that was used in Example 2 with an XML TRANSFORM FILE statement producing the file, **lant9b.xml**, and then transformed back into the original output format.

Next, the content of the XML document is imported from the file, **liant9c.xml**, and placed in the same data item using the XML IMPORT FILE statement.

Subsequently, the content of the XML document, **liant9c.xml**, is transformed using the internal XSLT stylesheet from the set of model files creating the file, **liant9d.xml**. This adds all of the COBOL attributes to **liant9d.xml**.

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

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

Data Item for Example 9

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

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "5914 West Courtyard Drive".
    02  Address-2  Pic X(64) Value "Suite 100".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78730.
    02  Time-Stamp Pic 9(8).
```

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

Other Definitions for Example 9

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example 9

The following tables show COBOL statements that relate to performing XML Extensions statements. Some COBOL statements (mostly the DISPLAY statements) have been omitted. The source of this example is in the file, **example9.cbl**.

Initialization (Example 9)

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 (Example 9)

COBOL Statement	Description
`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`  Liant-Address`
`  "liant9a"`
`  "Liant-Address".`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Transforming to External XML Format (Example 9)

COBOL Statement	Description

`XML TRANSFORM FILE`
`  "liant9a"`
`  "toext"`
`  "liant9b".`
	Execute the XML TRANSFORM FILE statement specifying:
	the input XML document filename,
	the XSLT stylesheet filename,
	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 (Example 9)

COBOL Statement	Description
`Move Spaces to Liant-Address.`	Ensure that the Liant-Address item contains no data.

`XML TRANSFORM FILE`
`  "liant9b"`
`  "toint"`
`  "liant9c".`
	Execute the XML TRANSFORM FILE statement specifying:
	the input XML document filename,
	the XSLT stylesheet filename,
	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 (Example 9)

COBOL Statement	Description

`XML IMPORT FILE`
`  Liant-Address`
`  "liant9c"`
`  "Liant-Address".`
	Execute the XML IMPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Transforming to Include COBOL Attributes (Example 9)

COBOL Statement	Description

`XML TRANSFORM FILE`
`  "liant9c"`
`  "example9"`
`  "liant9d".`
	Execute the XML TRANSFORM FILE statement specifying:
	the input XML document filename,
	the XSLT stylesheet filename,
	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 (Example 9)

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 (Example 9)

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

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

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

Status Display Logic (Example 9)

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

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

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

Execution Results for Example 9

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

COBOL Display (Example 9)

Note Pressing a key will terminate the program.

Running the program (**runcobol example9**) produces the following display:

```
Example-9 - Illustrate TRANSFORM FILE
liant9a.xml exported by XML EXPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16572692
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
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
16572692
liant9c.xml transformed into liant9d.xml by XML TRANSFORM FILE

You may inspect 'liant9a.xml' - 'liant9d.xml'

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

XML Documents (Example 9)

Microsoft Internet Explorer may be used to view the generated XML documents, **liant9a.xml**, **liant9b.xml**, **liant9c.xml**, and **liant9d.xml**. Their content of these documents should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

liant9a.xml – Internal Format (similar to liant1.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
- <liant-address xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-table/">
    <name>Liant Software Corporation</name>
    <address-1>5914 West Courtyard Drive</address-1>
    <address-2>Suite 100</address-2>
  - <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78730</zip>
    </address-3>
    <time-stamp>18413683</time-stamp>
  </liant-address>
```

liant9b.xml – External Format (similar to liant2.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
- <LiantAddress>
    <Information Name="Liant Software Corporation" Address1="5914 West
      Courtyard Drive" Address2="Suite 100" City="Austin" State="TX"
      Zip="78730" />
    <TimeStamp Value="18413683" />
  </LiantAddress>
```

liant9c.xml – Internal Format Restored

```
<?xml version="1.0" encoding="UTF-8" ?>
- <liant-address>
    <name>Liant Software Corporation</name>
    <address-1>5914 West Courtyard Drive</address-1>
    <address-2>Suite 100</address-2>
  - <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78730</zip>
    </address-3>
    <time-stamp>18413683</time-stamp>
  </liant-address>
```

liant9d.xml – Internal Format plus COBOL Attributes

```
<?xml version="1.0" encoding="UTF-8" ?>
- <root>
  - <liant-address version="1.0" cobtoxmlRevision="2.0"
      targetNamespace="http://tempuri.org/rmcobol/default/"
      targetIdRef="Q1_example-9" level="01" type="xsd:string" kind="GRP"
      length="239" offset="4" uid="Q1_example-9">
      <name level="02" type="xsd:string" kind="ANS" length="64" offset="4"
        uid="Q2_example-9">Liant Software Corporation</name>
      <address-1 level="02" type="xsd:string" kind="ANS" length="64"
        offset="68" uid="Q3_example-9">5914 West Courtyard
        Drive</address-1>
      <address-2 level="02" type="xsd:string" kind="ANS" length="64"
        offset="132" uid="Q4_example-9">Suite 100</address-2>
    - <address-3 level="02" type="xsd:string" kind="GRP" length="39"
        offset="196" uid="Q5_example-9">
        <city level="03" type="xsd:string" kind="ANS" length="32"
          offset="196" uid="Q6_example-9">Austin</city>
        <state level="03" type="xsd:string" kind="ANS" length="2"
          offset="228" uid="Q7_example-9">TX</state>
        <zip level="03" type="xsd:decimal" kind="NSU" length="5"
          offset="230" scale="0" precision="5" uid="Q8_example-
          9">78730</zip>
      </address-3>
      <time-stamp level="02" type="xsd:decimal" kind="NSU" length="8"
        offset="235" scale="0" precision="8" uid="Q9_example-
        9">18413683</time-stamp>
    </liant-address>
  </root>
```

Example A: Diagnostic Messages

This program illustrates the diagnostic messages that may be displayed for XML documents that are not well-formed or valid. The program uses the XML TEST WELLFORMED-FILE and XML VALIDATE FILE statements to test and validate a series of XML documents. (These predefined XML documents are detailed in the Program Description section.)

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML TEST WELLFORMED-FILE (on page 38), which verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE FILE (on page 40), which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example A

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example A

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

Line	Statement
1	`rmcobol examplea`
2	`slicexsy examplea Liant-Address -ss -bn`
3	`start /w runcobol examplea k`

Line 1 compiles the **examplea.cbl** source file with an embedded XML-format symbol table.

Line 2 builds the XML model files from the symbol table information in the RM/COBOL object program. By default, the model filenames are the same as the object filename with different extensions (in this instance, the examplea object filename is **examplea.cob** and the model filenames are **examplea.xml** and **examplea.xsd**). The option -ss produces a schema file, and the option -bn suppresses the banner message.

Line 3 executes **examplea.cob**. The K Option suppresses the runtime banner message. On line 3, the `start /w` sequence is included only as good programming practice.

Note The **examplea.xtl** file produced by **slicexsy** is not needed and could be deleted. The slicexsy utility is run for this example only to produce a schema and internal stylesheet to support the XML VALIDATE FILE statement in the example.

Program Description for Example A

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

In this example, three different predefined XML documents are processed:

- The **xlLianta1.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 for Example A

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

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "5914 West Courtyard Drive".
    02  Address-2  Pic X(64) Value "Suite 100".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78730.
    02  Time-Stamp Pic 9(8).
```

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

Other Definitions for Example A

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example A

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

Initialization (Example A)

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 (Example A)

COBOL Statement	Description
`XML TEST WELLFORMED-FILE "xlianta1".`	Execute the XML TEST WELLFORMED-FILE statement specifying the XML document filename.
`If Not XML-OK Perform Display-Status.`	If the statement terminates unsuccessfully, perform the Display-Status paragraph to display any error messages.
`XML TEST WELLFORMED-FILE "xlianta2".`	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 (Example A)

COBOL Statement	Description
`XML VALIDATE FILE "xlianta2" "examplea".`	Execute the XML VALIDATE FILE statement specifying: the XML document filename and the *ModelFileName#DataFileName* parameter value.
`If Not XML-OK Perform Display-Status.`	If the statement terminates unsuccessfully, perform the Display-Status paragraph to display any error messages.

Testing for a Well-Formed Document (Example A)

COBOL Statement	Description
`XML TEST WELLFORMED-FILE "xlianta3".`	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 (Example A)

COBOL Statement	Description
`XML VALIDATE FILE "xlianta3" "examplea".`	Execute the XML VALIDATE FILE statement specifying: the XML document filename and the *ModelFileName#DataFileName* parameter value.
`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example A)

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 (Example A)

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

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

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

Status Display Logic (Example A)

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

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

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

Execution Results for Example A

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

COBOL Display (Example A)

Running the program (**runcobol examplea**) produces three displays: the first is shown after the first diagnostic message, the second is shown after the second diagnostic message, and the third is displayed after some successful tests.

Note Because of differences in the underlying XML parsers, the results of running Example A vary between Windows and UNIX. When a parser error occurs, the current UNIX implementation does *not* display the offending line of XML text in error (as shown in the first display) because on UNIX systems, the underlying XML parser, libxml, does not support schema validation. Under UNIX, errors that would be detected by a schema are not reported (as illustrated in the second display). The third display, however, is the same under both implementations.

First Display

Note Pressing a key will cause the program to continue.

For Windows, the first display would be illustrated as:

```
Example-A - Illustrate diagnostics for invalid documents
 and documents that are not well-formed
XML TEST WELLFORMED-FILE - not well-formed
Error: 28[24] - in function: LoadDocument
Called from line 585 in EXAMPLEA(E:\xmlexample\EXAMPLEA.COB), compil\
ed 2008/08/18 18:08:49.
The name in the end tag of the element must match the element type in the start\
 tag.
line 2, position 261
<root><liant-address><name>Liant Software Corporation</name><address-1>5914 Wes\
t Courtyard Drive</address-1><address-2>Suite 100</address-2><address-3><city>A\
ustin</city><state>TX</state><zip>78730</zip></address-3><time-stamp>14525751</\
time-stamp></rm-address></root>
--------------------------------------------------------------------------------\
--------------------------------------------------------------------------------\
--------------------------------------------------------------------------------\
----------------------|
C:\liant\svnw2\trunk\rmc85\lixml\xml\xlianta1.xml
HRESULT: 0x80004005
Press a key to continue:
```

For UNIX, the first display would be shown as follows:

```
Example-A - Illustrate diagnostics for invalid documents
 and documents that are not well-formed
XML TEST WELLFORMED-FILE - not well-formed
Error: 28[2] - in function: LoadDocument
Called from line 585 in EXAMPLEA(/usr/xmltk/examples/examplea.cob), comp\
iled 2008/08/15 10:08:25.
/usr/xmltk/examples/xlianta1.xml
/usr/xmltk/examples/xlianta1.xml:2: parser error : Opening and endin\
g tag mismatch: liant-address line 2 and rm-address
state><zip>78730</zip></address-3><time-stamp>14525751</time-stamp></rm-address\
>
                                                                                \
^
Press a key to continue:
```

Second Display

Note Pressing a key will cause the program to continue.

For Windows, the second display would be illustrated as:

```
XML TEST WELLFORMED-FILE - well-formed - invalid
XML VALIDATE FILE - well-formed - invalid
Error: 28[24] - in function: LoadDocument
Called from line 598 in EXAMPLEA(C:\Liant\XMLExt\Examples\EXAMPLEA.COB), compil\
ed 2008/08/18 18:08:49.
Error parsing 'ABCDE' as decimal datatype.  The element 'zip' with value 'ABCDE\
' failed to parse.
line 10, position 127
   <zip level="03" type="xsd:decimal" kind="NSU" length="5" offset="230" scale=\
"0" precision="5" uid="Q8 example-a">ABCDE</zip>
--------------------------------------------------------------------------------\
-----------------------------------------------|
C:\DOCUME~1\BSINCL~1\LOCALS~1\Temp\T{720cde18-c03f-4196-824d-2ca57519b7a2}.xml
HRESULT: 0x80004005
Press a key to continue:
```

For UNIX, the second display would be shown as follows:

```
XML TEST WELLFORMED-FILE - well-formed - invalid
XML VALIDATE FILE - well-formed - invalid
Press a key to continue:
```

Third Display

Note Pressing a key will terminate the program.

```
XML TEST WELLFORMED-FILE - well-formed - valid
XML VALIDATE FILE - well-formed - valid
Status: +0000
Press a key to terminate:
```

For UNIX, the third display would be the same, but the XML document has only been verified to be well-formed and might not conform to the schema; however, for this example, the document does conform to the schema.

Example B: Import File with Missing Intermediate Parent Names

This COBOL program illustrates how an XML document with some missing intermediate parent names may be converted into COBOL data format and stored in a COBOL data item. (This capability of handling missing intermediate parent names has been included to make programs that deal with "flattened" data items, such as Web services, less complicated.) A COBOL program and an XML document file may contain the same elementary items, but may not have the identical structure. XML Extensions offers a way to handle such cases where there is not a one-to-one match between the COBOL data item and the XML document structure. Consider the following situation, in which the COBOL program imports a predefined XML document that has some missing intermediate parent names.

A missing intermediate parent name is an XML element name that corresponds to an intermediate-level COBOL group name. For example, in the following COBOL data item, the XML element name, `address-3`, is an intermediate parent name.

```
01  MY-ADDRESS.
    02  ADDRESS-1        PIC X(64) VALUE "101 Main St.".
    02  ADDRESS-2        PIC X(64) VALUE "Apt 2B".
    02  ADDRESS-3.
        03  CITY         PIC X(32) VALUE "Smallville".
        03  STATE        PIC X(2)  VALUE "KS".
```

The structure of the corresponding XML document would be:

```
  <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>
```

In cases where the intermediate parent name is not needed to resolve ambiguity, XML Extensions will attempt to reconstruct the document structure on input. For example, if the input XML document contained the following information, then the intermediate parent names of `address-3` and `my-address` would be added to produce an XML document compatible with the above document.

```
<root>
  <address-1>101 Main St.</address-1>
  <address-2>Apt 2B</address-2>
  <city>Smallville</city>
  <state>KS</state>
<root>
```

Example B illustrates this situation more fully.

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE (on page 34), which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Development for Example B

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example B

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

Line	Statement
1	`rmcobol exampleb`
2	`start /w runcobol exampleb k`

Line 1 compiles the **exampleb.cbl** source file with an embedded XML-format symbol table.

Line 2 executes **exampleb.cob**. The K Option suppresses the runtime banner message. On line 2, the `start /w` sequence is included only as good programming practice.

Program Description for Example B

This COBOL program illustrates how an XML document with some missing intermediate parent names may be converted into COBOL data format and stored in a COBOL data item.

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

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

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

Additionally, the content of the predefined XML document named **xliantb.xml**, which has some missing intermediate parent names, is also imported using the XML IMPORT FILE statement.

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

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

Data Item for Example B

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

```
01  Liant-Address.
    02  Name       Pic X(64) Value "Liant Software Corporation".
    02  Address-1  Pic X(64) Value "5914 West Courtyard Drive".
    02  Address-2  Pic X(64) Value "Suite 100".
    02  Address-3.
        03  City   Pic X(32) Value "Austin".
        03  State  Pic X(2) Value "TX".
        03  Zip    Pic 9(5) Value 78730.
    02  Time-Stamp Pic 9(8).
```

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

Other Definitions for Example B

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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).
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example B

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

Initialization (Example B)

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 (Example B)

COBOL Statement	Description
`Accept Time-Stamp From Time.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`  Liant-Address`
`  "liantb"`
`  "Liant-Address".`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Importing the Generated XML Document (Example B)

COBOL Statement	Description

`XML IMPORT FILE`
`  Liant-Address`
`  "liantb"`
`  "Liant-Address".`
	Execute the XML IMPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Importing the Predefined XML Document (Example B)

COBOL Statement	Description
`Initialize Liant-Address`	Ensure that the Liant-Address item is initialized.

`XML IMPORT FILE`
`  Liant-Address`
`  "xliantb"`
`  "Liant-Address".`
	Execute the XML IMPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	and the *ModelFileName#DataFileName* parameter value.

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example B)

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 (Example B)

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

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

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

Status Display Logic (Example B)

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

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

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

Execution Results for Example B

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

COBOL Display (Example B)

Note Pressing a key will terminate the program.

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

```
Example-B - Illustrate IMPORT with missing intermediate names
liantb.xml exported by XML EXPORT FILE
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
17010955
liantb.xml imported by XML IMPORT FILE:
Liant Software Corporation
5914 West Courtyard Drive
Suite 100
Austin                          TX78730
17010955
xliantb.xml imported by XML IMPORT FILE:
Wild Hair Corporation
8911 Hair Court
Sweet 4300
Lostin                          TX70707
00000000
You may inspect 'liantb.xml' & 'xliantb.xml'

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

XML Document (Example B)

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" ?>
- <liant-address xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-table/">
    <name>Liant Software Corporation</name>
    <address-1>5914 West Courtyard Drive</address-1>
    <address-2>Suite 100</address-2>
  - <address-3>
      <city>Austin</city>
      <state>TX</state>
      <zip>78730</zip>
    </address-3>
    <time-stamp>18414178</time-stamp>
  </liant-address>
```

xliantb.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
- <liant-address>
    <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>
  </liant-address>
```

Example C: Export File with Document Prefix

This program writes (or exports) an XML document file from the content of a COBOL data item. A document prefix is specified to declare entities referenced in the COBOL data item.

This example uses the following XML statements:

- XML INITIALIZE (on page 55), which initializes or opens a session with XML Extensions.
- XML EXPORT FILE (on page 28), which constructs an XML document (as a file) from the content of a COBOL data item.
- XML TERMINATE (on page 56), which terminates or closes the session with XML Extensions.

Note The XML EXPORT FILE statement contains an additional parameter: the name of the document prefix being used for the XML document export.

Development for Example C

The COBOL program must be compiled with an XML Extensions-enabled RM/COBOL compiler that generates and embeds an XML-format symbol table in the COBOL object file.

After the successful compilation, 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).

Batch File for Example C

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

Line	Statement
1	`rmcobol examplec`
2	`start /w runcobol examplec k`

Line 1 compiles the **examplec.cbl** source file with an embedded XML-format symbol table.

Line 2 executes **examplec.cob**. The K Option suppresses the runtime banner message. On line 2, the `start /w` sequence is included only as good programming practice.

Program Description for Example C

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

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

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

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

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

Data Item for Example C

The content of the COBOL data item `Line-Item` is as follows:

```
01  Line-Item.
    02 LI-BoilerPlate.
       03 FILLER      VALUE "&BoilerPlate;".
    02 LI-Name        PIC X(30).
    02 LI-Quantity    PIC 9(04).
    02 LI-CurPrice.
       03 FILLER      VALUE "&CURRENCY;".
       03 LI-Price    PIC 9(06)V99.
    02 LI-CurExt.
       03 FILLER      VALUE "&CURRENCY;".
       03 LI-Ext      PIC 9(10)V99.
```

This data item stores line item information, such as might appear in an invoice. There are three entity references, one to BoilerPlate and two to CURRENCY.

Document Prefix for Example C

The document prefix string is as follows:

```
78  DocumentPrefix    Value
      "<!DOCTYPE line-item [" & LF &
      "   <!ENTITY CURRENCY ""&#036;"">" & LF &
      "   <!ENTITY BoilerPlate ""All prices in USD"">" & LF &
      "       ]>".
```

This document prefix string provides a document type definition (DTD) that declares two entities in the internal subset. The first entity is CURRENCY, which is defined to be &#036, the “$” in UTF-8. The second entity is BoilerPlate, which is defined to be “All prices in USD”. The string will produce multiple lines in the exported document file because of the line feed characters introduced by the symbolic-character name LF.

Other Definitions for Example C

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

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

```
01  XML-data-group.
    03  XML-Status             PIC S9(4) Sign Leading Separate.
        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). .
    03  XML-COBOL-Version      PIC 9(4) VALUE 12. *>Used by XMLSetVersion
    03  XML-XMLIF-Version      PIC 9(4) VALUE 0.  *>Set by XMLSetVersion
```

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

Program Structure for Example C

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

Initialization (Example C)

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 (Example C)

COBOL Statement	Description

`XML EXPORT FILE`
`  Line-Item`
`  "liantc"`
`  "examplec"`
`  OMITTED  *> no stylesheet`
`  DocumentPrefix.`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,
	the XML document filename,
	the *ModelFileName#DataFileName* parameter value,
	the XSLT stylesheet (OMITTED),
	and the XML document prefix (DTD).

`If Not XML-OK Go To Z.`	If the statement terminates unsuccessfully, go to the termination logic.

Program Exit Logic (Example C)

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 (Example C)

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

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

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

Status Display Logic (Example C)

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

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

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

Execution Results for Example C

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

COBOL Display (Example C)

Note Pressing a key will terminate the program.

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

```
Example-C - Illustrate EXPORT FILE with Document Prefix
liantc.xml exported by XML EXPORT FILE

You may inspect 'liantc.xml'

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

XML Document (Example C)

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

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE line-item (View Source for full doctype...)>
- <line-item xmlns:xtk="http://liant.com/xcentrisity/xml-
  extensions/symbol-table/">
   <li-boilerplate>All prices in USD</li-boilerplate>
   <li-name>Widget</li-name>
   <li-quantity>50</li-quantity>
  - <li-curprice>
     $
     <li-price>5.42</li-price>
   </li-curprice>
  - <li-curext>
     $
     <li-ext>271</li-ext>
   </li-curext>
 </line-item>
```

In this display, the BoilerPlate entity reference is replaced with the DTD declared value “All prices in USD” and the CURRENCY entity references are replaced with the DTD declared value “$” (&#036 in UTF-8). If you view the source, the original entity references will be shown instead.

Example Batch Files

Three batch files are provided to facilitate use of the example programs: **cleanup.bat**, **example.bat**, and **examples.bat**.

cleanup.bat

This batch file will remove various files that were created by executing the example programs. This file contains a series of delete file commands similar to the following:

```
@echo off
echo cleanup started ...
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
echo                      finished cleanup.
```

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

```
cleanup
```

Note On UNIX systems, the script named **cleanup** is provided for the same purpose as **cleanup.bat** on Windows.

example.bat

This batch file will compile a COBOL source program, run the **slicexsy** utility against the compiled object code if model files are required, and finally execute the COBOL program. The content of this file is as follows:

```
@echo off
REM %1 == example program file name (without extension)
REM %2 == example data-item (when slicexsy needed)
REM %3 == slicexsy options (when slicexsy needed)
REM Compile the example program given by %1
rmcobol %1 k
REM Run slicexsy when schema/stylesheet model files required
if "%3" == "-ss"slicexsy %1 %2 %3 -bn
REM Run example program
start /w runcobol %1 a='\' 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 **slicexsy** utility will construct model files. The third parameter is used for passing options to the **slicexsy** utility.

To build and run Example 1: Export File and Import File (on page 90) using this batch file, enter the following on the command line:

```
example example1 Liant-Address -sn
```

Note On UNIX systems, the script named **example** is provided for the same purpose as **example.bat** on Windows.

examples.bat

This batch file will clean up files that were created from a previous run and then compile and run each example. The content of this file is similar to the following:

```
@echo off
call cleanup
echo Example1 - Export / Import File.
call example example1 Liant-Address -sn
echo Example2 - Export / Import with XSLT stylesheets.
call example example2 Liant-Address -sn
echo Example3 - Export / Import with Occurs Depending.
call example example3 Liant-Address -sn
echo Example4 - Export / Import with sparse arrays.
call example example4 Data-Table –sn
echo Example5 - Export / Import Text.
call example example5 Liant-Address -sn
echo Example6 - Export / Import with directory polling.
mkdir Stamp
call example example6 Time-Stamp -sn
echo Example7 - Export / Well-Formed File / Validate File.
call example example7 Liant-Address -ss
echo Example8 - Export / Well-Formed Text / Validate Text.
call example example8 Liant-Address -ss
echo Example9 - Export / Transform / Import.
call example example9 Liant-Address -ss
echo ExampleA - Well-Formed / Validate diagnostics.
call example examplea Liant-Address -ss
echo ExampleA - Import with missing intermediate names.
call example exampleb Liant-Address -sn
echo ExampleC - Export with document prefix.
call example examplec Line-Item -sn
```

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

```
examples
```

Note On UNIX systems, the script named **examples** is provided for the same purpose as **examples.bat** on Windows.

Appendix B: XML Extensions Sample Application Programs

XML Extensions provides several complete and useful sample application programs. The purpose of these self-contained programs is to demonstrate and explain how to perform typical application-building tasks in XML Extensions within a realistic context so that you can better see how to integrate them into your own applications.

Accessing the Sample Application Programs

The sample application programs are included in the XML Extensions samples directory, **Samples**.

Each sample application program is intended to reside in a separate subdirectory. For example, the XFORM sample resides in the directory named **Samples/xform**. Documentation for the sample is contained in the directory in the form of an HTML file.

On Windows systems, each application is packaged as a self-extracting executable program. For example, the XFORM sample is contained in the file **Samples/xform/xform.exe**. Running this application will extract its component parts. For the XFORM sample, this will produce the files, **xform.cbl** and **xform.htm**.

On UNIX systems, the applications were extracted when the samples were installed. The XFORM sample in **Samples/xform** contains the files, **xform.cbl** and **xform.htm**.

Appendix C: XML Extensions Error Messages

This appendix lists and describes the messages that can be generated during the use of XML Extensions.

Error Message Format

XML Extensions error messages may be several lines long. The general format of an error message includes the text of the message, and, if available, the COBOL traceback information, the name of the file or data item, and the parser information.

Note See Table 1 on page 181 for a summary of error messages.

Message Text

The first line of the error message has the following format:

```
<severity> - <message number> <message text>
```

severity indicates the gravity and type of message: Informative, Warning, or Error.

Message number is the documented message number followed by an internal message number in bracket characters. The internal number provides information for Liant technical support to use in diagnosing problems.

Message text is a brief explanation for the cause of the error.

An example of the first line of an error message is shown below:

```
Error: 28[12] - in function: LoadDocument
```

COBOL Traceback Information

The second line of the error message, present if the information is available, contains COBOL traceback information such as the following:

```
Called from line 421 in TEST15.COB(C:\DEV\TEST15.COB),
compiled 2003/05/14 09:42:06.
```

The error-reporting facility will try to break up lines that are too long for the line buffer provided in the COBOL program. This prevents long lines from being truncated. A backward slash character (\) is placed in the last position of the buffer and the line is continued on the subsequent line. For example, the traceback line shown above may be broken up as follows:

```
Called from line 421 in TEST15.COB(C:\DEV\TEST15.COB), co\
mpiled 2003/05/14 09:42:06.
```

Filename or Data Item in Error

The third line of the error message, present if the information is available, normally contains the name of the file or data item in error being referenced.

Parser Information

Note This section applies to the Windows implementation of XML Extensions only.

Additional lines may be present that contain parser or schema diagnostics from the underlying XML parser, such as:

```
Error parsing 'a9' as number datatype.
line 5, position 25
<ItemCount>a9</ItemCount>
-----------           --|
```

The first line of parser or schema diagnostic information contains an error message. The second line contains the line number and column position within the XML document. The third line contains the line of XML text in error. The fourth line contains an indicator that draws attention to the column position.

Summary of Error Messages

Table 1 describes the messages that may be generated when an error occurs in XML Extensions.

Table 1: XML Extensions Error Messages

Message Number	Severity and Message Text	Description
-4	Warning – subscript out of range	A subscript is out of range on import. The offending item is not imported.
-3	Warning – data truncation	A nonnumeric import item has been truncated to fit the associated COBOL data item.
-2	Warning – extraneous element	The import data contains one or more elements that do not belong anywhere in the COBOL data structure; the element(s) have been ignored.
0	Success	A normal completion occurred. No informative, warning, or error message was detected.
1	Informative – directory contains no documents	An XML FIND FILE statement did not find any XML documents (files with an .xml extension) in the specified directory.
2	Informative – document file – no data	An XML EXPORT FILE or an 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 slicexsy utility detected an invalid command line option. The option is ignored and processing continues.
5	Error – COBOL object file – invalid format	The slicexsy utility 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 slicexsy utility detected an error while attempting to open the specified COBOL object file.
7	Error – COBOL object file – read failure	The slicexsy utility detected an error while attempting to read data from the specified COBOL object file.
8	Error – COBOL object file – seek failure	The slicexsy 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 slicexsy utility or the xmlif library.

Table 1: XML Extensions Error Messages (*Cont.*)

Message Number	Severity and Message Text	Description
10	Error – cannot create URL	The xmlif library detected that a URL (a string beginning with the sequence http://, https://, or "file://") was used as an output document name.
11	Error – data item – duplicate found	The slicexsy utility detected more than one occurrence of the specified data item name in the COBOL object file or library. **Note** This message is valid only for versions of XML Extensions prior to version 12.
12	Error – data item – not found	The slicexsy utility 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 failed. This error may occur in the xmlif library or the slicexsy 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 detected an extra occurrence of a scalar data element. **Note** This message is valid only for versions of XML Extensions prior to version 12.
16	Error – example file – create failure	The slicexsy 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 slicexsy 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 slicexsy utility or the xmlif library.

Table 1: XML Extensions Error Messages (*Cont.*)

Message Number	Severity and Message Text	Description
23	Error – internal logic – node not found	The xmlif library 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 slicexsy utility to fail with this error while attempting to generate an XSLT stylesheet or schema.
25	Error – invalid data address	The xmlif library detected that the data structure address specified in an XML IMPORT or an 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 slicexsy utility was not re-executed to regenerate the model files.
26	Error – invalid object time stamp	While attempting to execute an XML IMPORT or an EXPORT statement, the xmlif library 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. **Note** This message is valid only for versions of XML Extensions prior to version 11.
27	Error – license management	The license verification logic in the slicexsy utility detected an error. **Note** This message is valid only for versions of XML Extensions prior to version 12.
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 slicexsy utility. Occasionally, XML Extensions generates documents that are then loaded as input documents. In the unlikely event that the generated document contains errors, a load document error will be encountered.

Table 1: XML Extensions Error Messages (*Cont.*)

Message Number	Severity and 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) or that the schema itself is in error. This error may occur in either the xmlif library or the slicexsy utility.
30	Error - in function: LoadStyleSheet	An error was detected while trying to load an internal or external XSLT stylesheet. This normally means that there was a problem locating the document (either the document does not exist or there is a problem with permissions). Another possible cause is that the XML parser software is not properly installed. This error may occur in either the xmlif library or the slicexsy utility.
32	Error - in function: LoadTemplate	An error was detected while trying to load an XML template file. This normally means that there was a problem locating the document (either the document does not exist or there is a problem with permissions). Another possible cause is that the XML parser software is not properly installed. This error may occur in the xmlif library.
33	Error - parameter - COBOL object file name missing	The slicexsy utility detected that the COBOL object file name command-line parameter is missing.
34	Error - parameter - data item name missing	The slicexsy utility detected that the name of the data item command-line parameter is missing.
35	Error - subscript out of range	While executing an XML IMPORT FILE or an XML IMPORT TEXT statement, the xmlif library 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). This error occurs only when the COBOL program was compiled with pre-version 12 copy files, that is, when a pre-version 12 application is using the version 12 XML Extensions. Applications compiled with the version 12 XML Extensions copy files return warning -4 instead.
36	Error - temporary file access error	The xmlif library encountered an 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.

Table 1: XML Extensions Error Messages (*Cont.*)

Message Number	Severity and Message Text	Description
37	Error - in function: TransformDOM	An unexpected error occurred while performing an XSLT transform of an XML document. This might be an internal error, but can be caused by an error in an external stylesheet. This error may occur in either the xmlif library or the slicexsy utility.
38	Error - in function: TransformText	An error occurred while performing an XSLT transform of an XML document using an external (user-supplied) XSLT stylesheet. This error may occur in the xmlif library.
39	Error - symbol table - not present in COBOL object	Version 11 and earlier meaning: The 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. Version 12 meaning: The RM/COBOL compiler did not produce an XML-format symbol table (either because it was not licensed to do so, or because the feature was disabled, for example, by the compiler configuration keyword SUPPRESS-XML-SYMBOL-TABLE being specified for the compilation). This error can be detected by either slicexsy or the xmlif library.
41	Error - old runtime version	The RM/COBOL runtime system is too old for this version of the xmlif library. Normally, this error cannot occur because the xmlif library will fail to load successfully when the runtime version is too low.
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 slicexsy utility.
43	Error - wrong COBOL object symbol table version	The slicexsy utility determined that the COBOL object symbol table version in the specified object file is newer than was available when this version of XML Extensions was released and, therefore, may contain features that are not supported by XML Extensions. Check with Liant Software for updates to XML Extensions. **Note** This message is not used in XML Extensions version 12 and later.
44	Error - wrong cobtoxml revision	The xmlif library determined that the format of the model files may be incompatible with the xmlif library. The template file is not version 1.0 or version 2.0, as required.

Table 1: XML Extensions Error Messages (*Cont.*)

Message Number	Severity and Message Text	Description
45	Error - invalid encoding selection	The value of the *Encoding* parameter of the XML SET ENCODING statement was neither "local" nor "utf-8".
46	Error - invalid UTF-8 data	An XML export operation failed because the data supplied was not valid for UTF-8.
47	Error - invalid RM_ENCODING value	The value of the RM_ENCODING environment variable on UNIX is not any of "rmlatin1", "rmlatin9", or a name recognized by the available iconv library.
48	Error - unable to locate iconv library	The value of the RM_ENCODING environment variable on UNIX is neither "rmlatin1" nor "rmlatin9", but an iconv library for character conversions could not be found.
49	Error - directory open failure	The XML FIND FILE statement was not able to locate and open the specified directory.
50	Error - missing XML parser (MSXML6)	The XML parser could not be found. This error occurs only on Windows and indicates the MSXML 6.0 parser is not installed in the Windows system. The MSXML 6.0 parser is normally installed when RM/COBOL is installed on Windows, but could not be found. MSXML 6.0 can be obtained by downloading it from Microsoft's web site or re-installing RM/COBOL.
51	Error - data item - illegal name format	A data-name being exported does not begin with an initial name character (letter, colon or underscore). This error should occur only for version 1.0 template files when a COBOL data-name begins with a digit. For version 2.0 template files, COBOL data-names that begin with a digit are prefixed with an underscore.
52	Error - CodeBridge conversion failure	The attempted conversion of data to or from a COBOL data type and a displayable string acceptable to XML failed.
53	Error - Name specified is not a data-item	When specifying a data structure name, a name was provided that is not a data item. For example, it might be a file-name, an index-name, a constant-name, and so forth. A data item name is required.
56	Error - can't inflate xml symbol table extracted from the object file	An error occurred while trying to inflate the extracted XML symbol table from the COBOL object file. There may be insufficient memory or disk space for the inflation, or the object file may have been corrupted.

Table 1: XML Extensions Error Messages (*Cont.*)

Message Number	Severity and Message Text	Description
57	Error - failed to create XML symbol table file for writing	An attempt to create an intermediate XML symbol table file for writing failed. This is frequently an issue with permissions on the directory containing the temporary file.
60	Error - XML symbol table file – read failure	An error occurred while trying to process the XML-format symbol table.
62	Error - requested template file cannot be found	Requested template file specified by the model file data name parameter cannot be found.
63	Error - resolved file name is too long	The resolved filename from one of the following statements is too large to fit in the buffer provided: XML EXPORT FILE XML EXPORT TEXT XML IMPORT FILE XML IMPORT TEXT XML RESOLVE DOCUMENT-FILE XML RESOLVE SCHEMA-FILE XML RESOLVE STYLESHEET-FILE XML RESOLVE MODEL-FILE
64	Error - resolved file name does not exist	The file name passed to one of the following statements cannot be resolved. It may not exist and is not accessible. Check the value of the RUNPATH environment variable to verify that all search paths have been specified. XML EXPORT FILE XML EXPORT TEXT XML IMPORT FILE XML IMPORT TEXT XML RESOLVE DOCUMENT-FILE XML RESOLVE SCHEMA-FILE XML RESOLVE STYLESHEET-FILE
65	Error - name / value pair required	The XML SET XSL-PARAMETERS statement requires an even number of parameters (name / value pairs).
66	Error - excessive number of XSL parameters	The XML SET XSL-PARAMETERS statement is limited to a maximum of 40 parameters (20 name / value pairs).
67	Error - unique identifier too long for buffer	The buffer supplied in the XML GET UNIQUEID statement is too small. Unique identifiers require 38 character positions.

Appendix D: slicexsy Utility Reference

This appendix describes the optional **slicexsy** utility.

What is the slicexsy Utility?

The **slicexsy** utility is an optional application program that has been provided for backward compatibility with previous versions of XML Extensions. It also allows for schema validation.

The **slicexsy** utility provides an alternative to deploying object programs that include the entire XML-format symbol table produced by the RM/COBOL compiler licensed for XML Extensions. This symbol table may be too large to be loaded by XML Extensions on some platforms. In such cases, the **slicexsy** utility may be used to produce a "slice," or subset, of the XML-format symbol table that is smaller and easier to load on the deployment machine.

The **slicexsy** utility produces a set of three XML Extensions-deployable files that are known as model files (see page 196). Model files describe a single data structure within the COBOL program.

- If schema validation is to be performed, all three model files (.**xtl**, **.xsl**, and **.xsd**) must be deployed.
- If schema validation is not performed, it is necessary to deploy only the template file (the model file having the .**xtl** extension).

By default, the **slicexsy** utility does not produce schema information.

Things to Consider Before Using slicexsy

XML Extensions does not require the use of the **slicexsy** utility. Developers may wish to consider the advantages and disadvantages of doing so prior to employing the utility as a deployment tool.

The disadvantages of using **slicexsy** include the following:

- It is possible to have out-of-date model files that do not match the currently running program, which can result in odd failures that are difficult to debug.
- Because **slicexsy** is a separate program that must be run after compilation, it adds an extra step to the development cycle.
- Furthermore, not using **slicexsy** also simplifies the deployment process in that there are fewer files to deploy.

Those considerations aside, however, the **slicexsy** utility provides a number of advantages:

- It allows the developer to validate data using a schema.
- It allows the developer to reduce the size of the object programs that are deployed.
- It enables faster XML loading.
- It is less resource-intensive on deployment machines.
- It provides additional security. By eliminating the symbol table, access to a map of the developer's data structures is also eliminated.

Using the slicexsy Utility

To use the **slicexsy** utility, you specify (at a minimum) the name of a COBOL object file and the name of a COBOL data item within that file. If the application wishes to use several COBOL data structures as separate XML documents within the same COBOL application, it is necessary to run the **slicexsy** utility once for each data structure, using an optional parameter to provide a name for the model files.

The **slicexsy** utility requires that the COBOL object program be compiled in such a manner that an XML-format symbol table is embedded in the COBOL object file. The generation of an XML-format symbol table is controlled by whether or not the RM/COBOL compiler is licensed for XML Extensions and also by the following configuration file option:

```
COMPILER-OPTIONS SUPPRESS-XML-SYMBOL-TABLE=<value>
```

where, <*value*> may be YES or NO. The default is NO, resulting in the production of the XML-format symbol table by default.

Note An RM/COBOL compiler that is not licensed for XML Extensions will not produce an XML-format symbol table regardless of the setting of this configuration keyword.

Once the **slicexsy** utility creates a template file based on the XML-format symbol table, the symbol table may then be removed from the deployed object programs. To remove the symbol table, the source program must be recompiled using the SUPPRESS-XML-SYMBOL-TABLE keyword with the value set to YES.

File Naming Conventions

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

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

Model File Naming Conventions

Model files, the XML documents generated by the **slicexsy** utility, have predetermined extensions. If configured to do so, the **slicexsy** utility generates a set of three files from a single filename with different extensions. A set of model files consist of the following:

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

  Note On UNIX systems, the underlying XML parser, **libxml**, does not support schema validation

For a more detailed discussion, see Model Files (on page 196).

Backward Compatibility

The **slicexsy** utility was introduced in version 12 of XML Extensions as a replacement for the **cobtoxml** utility. To accommodate batch streams created with earlier versions that use **cobtoxml**, XML Extensions allows **slicexsy** to be referenced as **cobtoxml**. This can be done by copying or renaming **slicexsy** (or **slicexsy.exe**) to **cobtoxml** (or **cobtoxml.exe**) or by using the UNIX **ln** command to link a new name (**cobtoxml**) to the existing name (**slicexsy**).

It should be noted that regardless of whether it is named **slicexsy** or referenced as **cobtoxml**, this utility works only with the XML-format symbol table introduced in version 12 of XML Extensions. Prior to version 12, the **cobtoxml** utility required that the COBOL object program be compiled with the RM/COBOL Compile Command Y Option enabled in order to place the debug symbol table information in the object file; this is not a requirement in version 12.

The name used to invoke the utility determines the default for the schema option when no schema option is specified: **slicexsy** assumes **–sn** (no schema) and **cobtoxml** assumes **–ss** (schema XSD). When a schema file is produced, a stylesheet is also produced in addition to the schema and the template.

It should be noted that the command line name (**–n[afhlmpu]**) and alternative schema options (**–sb** and **–sd**), which were available with the original **cobtoxml** utility, are not supported by the **slicexsy** utility no matter what name is used to invoke the utility.

When the utility is referenced as **cobtoxml**, the usage screen appears as follows as a help screen:

```
RM/COBOL Symbol Table Slice Utility
 Version nn.nn for operating-system-name.
 Copyright (c) 2008 by Liant Software Corp.  All rights reserved.

Usage: cobtoxml cob-file-name data-item-name model-file-name options

  cob-file-name: case-sensitive name of the RM/COBOL object file
  data-item-name: case-insensitive name of the COBOL data item
  model-file-name: optional case-sensitive name for the XML file(s)

  If model-file-name omitted, it defaults to cob-file-name.

  Options:

      -bc -- banner:  copyright only (default)
      -bn -- banner:  none
      -bv -- banner:  verbose

      -sn -- schema none
      -ss -- schema xsd (default)

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

Command Line Interface

The **slicexsy** utility (**slicexsy.exe** on Windows and **slicexsy** on UNIX) is executed with either of the following command syntax formats:

Syntax Format 1

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

Syntax Format 2

```
slicexsy cob-file-name#data-item-name [model-file-name] [options]
```

Notes

- Syntax Format 1 can be used to mimic the behavior of the utility prior to version 12 when it was named **cobtoxml**. If the optional *model-file-name* parameter is omitted, Syntax Format 1 causes the value of *cob-file-name* to be used as a base for model files, which is compatible with **cobtoxml** behavior. For example, the following command

  ```
  slicexsy myfile mydata
  ```

  would generate model files based on myfile (**myfile.xtl**, and so forth).

- Syntax Format 2 is the behavior introduced in version 12 with the **slicexsy** utility. If the optional *model-file-name* parameter is omitted, Syntax Format 2 causes the value of *data-item-name* to be used as a base for model files. For example, the following command:

  ```
  slicexsy myfile#mydata
  ```

  would generate model files based on mydata (**mydata.xtl**, and so forth).

cob-file-name, the first positional input parameter, is the case-sensitive name of the RM/COBOL object file that includes an XML-format symbol table. The generation of this XML-format symbol table is controlled by the configuration file option, COMPILER-OPTIONS SUPPRESS-XML-SYMBOL-TABLE. If this parameter contains an extension, it will be used as entered. If the extension is omitted, **.cob** will be added. If the file specified by *cob-file-name* does not exist in the current directory, the RUNPATH environment variable will be used to search for the file .

data-item-name, the second positional input parameter, is the case-insensitive name of the selected data item within a COBOL program. Qualification for uniqueness of reference may be required. The most common use is the name of a record-name (level-number 01), but a group at any level or an elementary data item may be referenced. A *data-item-name* must be unique in all programs within the object file (all separately compiled programs, in the case of program libraries). In order to achieve uniqueness, the name may be qualified using the "//" sequence, which is similar to the XPATH specification format. That is, the **slicexsy** utility specifies qualifiers in the form "A//B//C" or A//C, which corresponds to the COBOL qualification C OF B OF A and C OF A, respectively. In the case of program libraries, all separately compiled programs are searched. The leading qualifiers in the *data-item-name* parameter may be program-names in order to achieve uniqueness when the same data-name occurs in multiple programs in the program library. When the *data-item-name* is used as the basis for the *model-file-name(s)*, the last component of a qualified name is used. For example:

```
slicexsy myfile#item-1//item-2
```

would cause the model file

```
item-2.xtl
```

to be generated.

model-file-name, the optional third positional output parameter, is the name of the set of XML documents, called model files, having a single base filename with different, predetermined extensions (**.xtl**, **.xsl** and **.xsd**) that are produced by the **slicexsy** utility and that describe the COBOL data item. The value of this name is treated as case sensitive. If this parameter already contains an extension, it will be ignored. For more information, see Model Files (on page 196). On Windows, either a forward slash "/" or a backward slash "\" character may be used as a directory-separator character when specifying a filename. This may make the representation of a Syntax Format 2 *cob-file-name#data-item-name* to be familiar to the developer. For example:

```
slicexsy code/myfile#structure//name
```

instead of

```
slicexsy code\myfile#structure//name
```

options represents command line options, which are described in Command Line Options (on page 195). Although this parameter is shown as the last parameter, it may occur anywhere after **slicexsy** on the command line. Additionally, *options* may be specified multiple times. If contradictory options are selected (such as, **–ss –sn**), the last option selected is used. Invalid options display a diagnostic message. Option letters are case insensitive; that is, the following combinations are equivalent: "-bc", "-bC", "-Bc" and "-BC". The *options* parameter is divided into two categories: banner and schema.

Note When no command line parameters are entered, the following **slicexsy.exe** usage message is displayed as a help screen:

```
RM/COBOL Symbol Table Slice Utility
 Version nn.nn for operating-system-name.
 Copyright (c) 2008 by Liant Software Corp.  All rights reserved.

Usage: slicexsy cob-file-name data-item-name model-file-name options
   or: slicexsy cob-file-name#data-item-name model-file-name options

  cob-file-name: case-sensitive name of the RM/COBOL object file
  data-item-name: case-insensitive name of the COBOL data item
  model-file-name: optional case-sensitive name for the XML file(s)

  If model-file-name omitted, it defaults to cob-file-name unless
  "#" syntax used, in which case it defaults to data-item-name.

  Options:

      -bc -- banner:  copyright only (default)
      -bn -- banner:  none
      -bv -- banner:  verbose

      -sn -- schema none (default)
      -ss -- schema xsd

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

Command Line Options

The following options are available on the **slicexsy** command line: banner and schema.

Note Name options, which were available with the **cobtoxml** utility (version 9 of XML Extensions), are not supported by the **slicexsy** utility**.**

Banner Options

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

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

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

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

Schema Options

Note The UNIX version of XML Extensions does not support schema files.

If desired, a schema file can be 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**. Two formats of schema files are defined: Schema and None.

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

Supported schema options include the following:

Option	Description
–ss	The generated schema file complies with the standard schema definition.
–sn	No schema file is generated. (This is the default.)

Model Files

The **slicexsy** utility creates a set of three XML documents known as model files for each data structure that is specified within the COBOL program. Model files have the same root name as the object file, although each filename has a unique, predetermined extension. The following types of model files are created:

- Template file
- Internal XSLT stylesheet file
- Schema file

Note On UNIX systems, the underlying XML parser, **libxml**, does not support schema validation.

CAUTION Developers who use the **slicexsy** utility should be aware that programs may recompile without always running the **slicexsy** utility. It is necessary to run **slicexsy** *only when the specified data structure(s) are changed*. Therefore, it is the programmer's responsibility to specify the correct and current model file. Specifying an incorrect or non-current model file may result in the wrong data being exported or imported.

Template File

The template file is the most important of the model files, as it governs both import and export operations. XML Extensions uses the template file to generate an XML document that is a subset of the XML-format symbol table in the COBOL object program. A template file has the extension **.xtl**. Although the template file does not contain any text values, each element in the file contains several COBOL-like attributes that describe the data. These attributes provide the additional information XML Extensions 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:

```
new-price-1 (<dataItem level="03" name="new-price-1" type="xsd:string"
   kind="NSE" length="14" offset="4" uid="Q4_test-99" />)
```

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 `"xsd:string"`. This information tells XML Extensions to obtain data from the COBOL data structure and convert the data from COBOL data format to a text format for the XML document.

When the template file is generated with **slicexsy**, the file would normally be distributed with the application. The template file can be omitted since the template can be extracted from the object file at execution time. However, if the COBOL program is later compiled with the SUPPRESS-XML-SYMBOL-TABLE configuration keyword set to the value YES (for example, to reduce the size of the application), the template file is required as part of the application.

Internal XSLT Stylesheet File

The internal XSLT stylesheet is used in conjunction with the schema file by the XML VALIDATE FILE and XML VALIDATE TEXT statements to perform schema validation in order to ensure that the XML document conforms to XML syntax rules.

Schema File

In XML terminology, a "schema" is a set of rules that defines an XML document. It is a description of how data is structured, and it is *about* the data rather than the data itself. Although, by default, the **slicexsy** utility does not produce schema information, there are cases where validation by schema files may be appropriate. In such instances, the **slicexsy** utility has an option to generate a schema file, as described in Schema Options (on page 195). The schema file may be used to validate the content of an XML document, as detailed in XML VALIDATE FILE (on page 40) and XML VALIDATE TEXT (on page 41).

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. Because schema rules can be strict, it is sometimes difficult for a document to pass the validation. For example, XML requires that elements be properly nested. The sample code below is not well-formed XML, because the *"Em"* and *"Strong"* elements overlap:

```
<p>Normal <em>emphasized <strong>strong emphasized</em> strong</strong></p>
```

Beginning with version 12 of XML Extensions, however, the schema rules have been relaxed such that an imported document may be a subset of a valid document. The imported document may have missing parents. The following is an example that illustrates this situation: a) the underlying COBOL data structure, b) the schema file showing that the document defining this structure is well-formed, and c) the schema file showing a subset of the valid document that has missing parents.

a) COBOL data structure

```
01  A.
    03  B.
        05  C    PIC 9    VALUE 1.
```

b) XML schema file describing this data structure

```
<a>
    <b>
        <c>1</c>
    </b>
</a>
```

c) XML schema file with missing parents (b, the parent of c, is missing)

```
<a>
      <c>1</c>
</a>
```

The schema file (**.xsd**) and the internal XSLT stylesheet file (**.xsl**) are both used by the XML VALIDATE FILE and XML VALIDATE TEXT statements to perform schema validation, thus ensuring that the XML document conforms to XML syntax rules.

Referencing XML Model Files

XML model files (see page 196) may be referenced by the COBOL application by means of a traditional path name or by an Internet address. Examples of references to XML model files are shown in the following table.

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

Appendix E: Summary of Enhancements

This appendix provides a summary of the new features and changes in the various releases of XML Extensions, beginning with the most recent release.

Notes

- The information in this appendix is historical. It was accurate at the time written for the specific version being described. Various features may have changed in later releases, and, possibly, some features may have been removed or changed.
- Beginning with the version 9 release, the name of this product changed from "XML Toolkit for RM/COBOL" to "XML Extensions".

Version 12

This section summarizes the major enhancements available in version 12 of XML Extensions on Windows and UNIX. Many of these enhancements have also been distributed with various releases of Xcentrisity Business Information Server (BIS).

- An XML Extensions-enabled RM/COBOL compiler will generate and embed an XML-format symbol table in the COBOL object file that matches the currently-running program. This new method provides a number of benefits, including the following:
  - In previous versions of XML Extensions, it was possible to have out-of-date model files that did not match the current program. This situation sometimes resulted in odd failures that were difficult to debug.
  - The XML-format symbol table contains more information than the debugging symbol table used by previous versions of XML Extensions.
  - Because the XML-format symbol table is compressed in the object file, it serves to obscure information about the data layouts that were previously exposed in model files.
  - Both the development and deployment processes are simplified in that it is no longer necessary to run an additional program after compilation and there are fewer files to deploy.

- For backward compatibility, the **cobtoxml** utility has been replaced by a new utility named **slicexsy**. For more information, see Appendix D: *slicexsy Utility Reference* (on page 189).
- A schema file is no longer created by default and validation against a schema is no longer performed during import and export operations. If validation is required, the **slicexsy** utility must be directed to produce a schema file and separate calls must be made to either the XML VALIDATE FILE or XML VALIDATE TEXT statements. Schema validation has been relaxed to allow an imported document to have missing parents. For further details, see Document Processing Statements (on page 27).
- The *ModelFileName* parameter for the XML IMPORT FILE, XML IMPORT TEXT, XML EXPORT FILE, and XML EXPORT TEXT statements has been extended to a new format, *ModelFileName#DataFileName*, which allows the specification of either or both the filename and the data structure name. For detailed information, see the parameter definition in the statement descriptions in Chapter 3: *XML Extensions Statements Reference* (on page 25).
- The following new statements have been added:
  - The XML COBOL FILE-NAME statement allows the developer to set the *ModelFileName* (the string before the #) in the *ModelFileName#DataFileName* parameter of various subsequent XML Extensions statements. The default value will be used when the *ModelFileName* string is not specified in the *ModelFileName#DataFileName* parameter of those subsequent statements.
  - The XML RESOLVE DOCUMENT-NAME statement is used to resolve the name of an XML document file. The resolution process is the same as that for the *DocumentName* parameter of an XML IMPORT statement.
  - The XML RESOLVE SCHEMA-FILE statement is used to resolve the name of an XML schema file created using the optional **slicexsy** utility, as described in Appendix D: *slicexsy* Utility Reference (on page 189). The resolution process is similar to that for the *ModelFileName#DataFileName* parameter of an XML IMPORT or XML EXPORT statement. The value of this parameter must specify an existing template file (**.xtl** extension) and not a COBOL object file (**.cob** extension).
  - The XML RESOLVE STYLESHEET-FILE statement is used to resolve the name of an XML stylesheet file. The resolution process is the same as that for the *StyleSheetName* parameter of an XML IMPORT or XML EXPORT statement.
  - The XML RESOLVE MODEL-NAME statement is used to resolve the name of a model file/data name combination. The resolution process is the same as that for the *ModelFileName#DataFileName* parameter of the XML IMPORT FILE, XML IMPORT TEXT, XML EXPORT FILE, or XML EXPORT TEXT statements.
  - The XML COMPATIBILITY MODE statement allows version 12 of XML Extensions to be compatible with existing data and applications by inserting <root> as the top level entry in a document during an export operation. While versions of XML Extensions prior to version 12 required that <root> be the top level element of a document, version 12 and later of XML Extensions will tolerate either the presence or absence or the <root> element.
  - The XML GET FLAGS statement retrieves the setting of the flags that are used for internal data conversion. Valid flag values are specified in the copy file, **lixmldef.cpy**. The initial setting of the flags has the following flag values set: PF-Leading-Spaces, PF-Trailing-Spaces, PF-Leading-Minus, and PF-Rounded. The setting of the flags can be changed with the XML SET FLAGS statement. The XML GET FLAGS statement retrieves 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.

- The XML TRACE statement generates trace information to a designated file. The statement name and parameter values (as well as the calling program name and the time executed) are recorded on entry. Updated parameter values are displayed on exit.
- The XML SET XSL-PARAMETERS statement passes a list of name/value pairs to XML Extensions.
- The XML CLEAR XSL-PARAMETERS statement clears sets of name/value pairs that have been stored in XML Extensions by the XML SET XSL-PARAMETERS statement.

- Previously, only filenames that began with “http://” and “https://” were recognized as URLs. This URL recognition has been expanded to include “file://”.
- A newly revised second edition of the *XML Extensions User’s Guide* is now available. It serves as the base document for the XML Extensions component and covers version 12 (the current release).

Version 9

This section summarizes the major enhancements available in version 9 of XML Extensions on Windows and UNIX. Many of these enhancements have also been distributed with various releases of Xcentrisity Business Information Server (BIS).

- **RM/COBOL Object Version 12 Support.** XML Extensions now supports RM/COBOL object version 12, which was introduced with RM/COBOL version 9.
- **UNIX Diagnostics.** Better diagnostic information is returned when XML IMPORT FILE/TEXT statements, discussed in Document Processing Statements (on page 27), fail due to an XSLT transform error.
- **Windows XSLT Stylesheet Processing.** XSLT stylesheets that used a literal result element were incorrectly encoded in UTF-16 on Windows. The encoding for the literal result was fixed to be UTF-8.
- **Missing Windows MSXML Parser.** A more descriptive diagnostic is returned if Microsoft's MSXML 4.0 parser is not installed. For further details, see System Requirements for Windows (on page 7).
- **Buffer Overrun Problem.** The XML import statements now verify that input data will fit in selected data structure.
- **URL Recognition.** Previously, only filenames that began with "http://" were recognized as URLs. This has been expanded to include "https://".
- **Filename Extensions.** Normally, if a filename extension is not present, one is added. However, with URLs (especially on the Internet), the filename must be used exactly as it is specified. Consequently, the processing of filename extensions has been modified so that a filename extension is never added to a filename that is a URL.
- **RUNPATH Search.** The RUNPATH search sequence has been modified to ignore directory names that use the Universal Naming Convention (UNC) notation (for example, "//system/directory"). UNC names are normally used in an application that uses RM/InfoExpress. XML Extensions cannot access files directly through RM/InfoExpress. By ignoring UNC directory names, unnecessary time delays are avoided when

performing a RUNPATH search. For further information, see Automatic Search for Files (on page 67).

- **cobtoxml Banner.** The **cobtoxml** utility has been modified to display the banner when necessary command line parameters are omitted. For more information, see Command Line Interface (on page 192).
- **XML Export Blank Suppression.** In prior versions, the XML EXPORT FILE and EXPORT TEXT statements would strip leading spaces from all nonnumeric data items. Leading spaces are now stripped only from data items that are defined with the JUSTIFIED phrase. For more information, see Data Representation (on page 69) and Handling Spaces and Whitespace in XML (on page 86).
- **XSLT Stylesheets with DTD.** The loading of XSLT stylesheets has been improved to allow the stylesheet to contain a document type definition (DTD). Previously, the presence of a DTD in a stylesheet caused a validation error on load. A DTD is required if the stylesheet uses entity references that are not predefined by XML. Stylesheets with HTML or XHTML entity references, such as " " and "©" are often generated by commonly used stylesheet generator tools. The tool may not generate the DTD, so the DTD must be added manually after the XSLT stylesheet is generated. For more information, see Document Type Definition Support (on page 84).
- **Improved Namespace Support for Schema Validation.** The XML VALIDATE FILE and XML VALIDATE TEXT statements would fail in the LoadSchema function if the specified schema contained a `targetNamespace` attribute. This has been fixed so that the schema loads successfully and is properly referenced in the schema collection by the URL used as the value of the `targetNamespace` attribute of the schema.

Version 2

This section summarizes the major enhancements available in version 2 of XML Toolkit:

- **Support for UNIX.** XML Extensions is currently available for selected UNIX platforms, including AIX, HP-UX, Linux, SCO OpenServer, Sun Solaris, and UnixWare.

  The Windows implementation continues to use Microsoft's XML parser, MSXML 4.0 or greater. The UNIX implementation is based on the XML parser (libxml) and the XSLT transformation parser (libxslt) from the C libraries for the Gnome project. While the Windows implementation continues to support the use of schema files, the UNIX implementation of schema support in the underlying XML parser (libxml) is still under development.
- Document Type Definition Support (on page 84). Exporting of XML documents has been enhanced to include the ability to specify a document type definition, which defines the legal building blocks of an XML document. A DTD can be used to define entity names that are referred to by the values of FILLER data items in the COBOL data structure being exported.
- Anonymous COBOL Data Structures (on page 78). The acts of exporting and importing of documents have been improved so that an anonymous COBOL data structure can be used. An anonymous COBOL data structure is any data area that is the same size or larger than the data structure indicated by the template file. This means that exporting from and importing to a Linkage Section data item, which is either based on an argument passed to a called program or a pointer set by the SET statement (for example, into allocated memory), is now possible. This same capability is also true for an external data item.

- **Relaxed Time Stamp Checking.** It is no longer necessary for the compilation time stamp in the object program to match the **cobtoxml** time stamp in the template file. That is, the program may be recompiled without running the **cobtoxml** utility. It is necessary to run **cobtoxml** only when the relevant data structure(s) are changed.
- **UTF-8 Data Encoding.** Support has been added to both the UNIX and Windows implementations of XML Extensions to allow the in-memory representation of element content to use UTF-8 encoding. UTF-8 is a format for representing Unicode. This may be useful for COBOL applications that wish to pass UTF-8 encoded data to other processes. XML documents are normally encoded using Unicode. XML Extensions always exports XML documents with UTF-8 data encoding. For further information, see the applicable topics under Data Representation (on page 69) and the discussion of XML and Character Encoding (on page 83).
- **New XML Statement.** A new XML statement, XML SET ENCODING (on page 63), has been added to XML Extensions that allows the developer to switch between the local character encoding (which is system-dependent) and the UTF-8 encoding format.

Version 1

This was the initial release of the XML Toolkit. Version 1 of the XML Toolkit for RM/COBOL ran on Microsoft Windows 32-bit operating systems, excluding Windows 95.

The XML Toolkit for RM/COBOL is Liant Software Corporation's facility that allows RM/COBOL applications to access XML (Extensible Markup Language) documents. XML is the universal format for structured documents and data on the Web.

Glossary of Terms

The glossary explains the terminology used throughout XML Extensions.

Terminology and Definitions

The following terms are defined.

Array

A COBOL table, that is, a data item described with the OCCURS clause.

Caching

Caching is a means of increasing performance by keeping loaded XSLT stylesheets, templates, and schema documents in memory for reuse without the need to reload them. If the application dynamically generates new copies of such documents, caching may be permanently or selectively disabled by the application. Caching is enabled by default at the beginning of an application.

COBOL data structure

A COBOL data structure is a COBOL data item. In general, it is a group data item, but in some cases, it may be a single elementary data item. An XML Extensions-enabled RM/COBOL compiler generates and embeds an XML-format symbol table in the COBOL object file. The XML-format symbol table provides a map between the COBOL data structure specified in an XML Extensions statement and the XML representation of the COBOL data structure. This map can be used move data in either direction at runtime. Extensible Stylesheet Language Transformations (XSLT) of the XML data representation can be used to match XML element names to COBOL data-names in cases where the names differ.

Document Type Definition (DTD)

The document type definition occurs between the XML header and the first element of an XML document. It optionally declares the document structure and entities. Declared entities may be referenced in the document.

DOM

Acronym for Document Object Model. XML documents are parsed and stored in the DOM for processing.

External XSLT stylesheet

An XSLT stylesheet that is provided by the user and referenced as a parameter in the XML EXPORT FILE/TEXT, XML IMPORT FILE/TEXT, or XML TRANSFORM statements. (The term "external" is used in this document to differentiate, where necessary, between the model file called the "internal XSLT stylesheet" and user-supplied "external" XSLT stylesheets.) See also XSLT stylesheet (on page 209).

HTML

An acronym for Hypertext Markup Language. A text description language related to SGML; it mixes text format markup with plain text content to describe formatted text. HTML is ubiquitous as the source language for Web pages on the Internet. Starting with HTML 4.0, the Unicode Standard functions as the reference character set for HTML content. See also SGML (on page 206), XHTML (on page 208), and XML (on page 208).

iconv

A character conversion library available on some UNIX systems for converting between UNICODE characters and local characters. When an iconv library is available, the RM_ENCODING environment variable may specify the name of a conversion supported by that iconv library and the xmlif library will use that conversion. Otherwise, the only conversions supported are "rmlatin1" and "rmlatin9".

Internal XSLT stylesheet

An XSLT stylesheet that is one of the model files created by the **slicexsy** utility. The internal XSLT stylesheet is used in conjunction with the schema file by the XML VALIDATE FILE and XML VALIDATE TEXT statements to perform schema validation in order to ensure that the XML document conforms to XML syntax rules.

Model files

XML document files created by the **slicexsy** utility. These include the template (*modelname*.xtl), internal XSLT stylesheet (*modelname*.xls), and schema (*modelname*.xsd) files.

Schema valid XML document

An XML document that conforms to a particular XML schema.

SGML

An acronym for Standardized Generalized Markup Language. A standard framework, defined in ISO 8879, for defining particular text markup languages. The SGML framework allows for mixing structural tags that describe format with the plain text content of documents, so that fancy text can be fully described in a plain text stream of data. See also HTML (on page 206) and XML (on page 208).

Structured document

The term “structured document” describes the concept that a document can contain content, such as words, numbers, pictures, and so forth., as well as information describing the role of content elements and substructures. Adding “structure” to documents facilitates searching, sorting, or any one of a variety of operations to be performed on an electronic document. The benefits of adding structure to electronic documents include portability, re-usability, inter-system operability, ease-of-storage and retrieval, longevity, quick access, and low distribution costs. XML is a set of rules for structuring a document using hierarchical markup. See also XML (on page 208).

Stylesheet

See XSLT stylesheet (on page 209).

UNC

An acronym for Universal Naming Convention. UNC is a filename format that is used to specify the location of files, folders, and resources on a local are network (LAN). For example, a UNC address may look something like this:

\\server-name\directory\filename

UNC also can be used to identify peripheral devices shared on the network, including scanners and printers. It provides each shared resource with a unique address, which allows operating systems that support UNC (such as Windows) to access specific resources quickly and efficiently.

Unicode

Unicode was developed to support the worldwide interchange, processing, and display of diverse languages and technical disciplines of the world. Unicode is a character coding system that assigns a unique number to each character in each of the world’s principal written languages. There exist several alternatives for how a sequence of such characters or their respective integer values can be represented as a sequence of bytes. The two most obvious encodings store Unicode text as either 2- or 4-byte sequences. The official terms for these encodings are UCS-2 and UCS-4, respectively. The current version of the Unicode Standard, developed by the Unicode Consortium, is v4.0.0. For an alternative encoding of Unicode, see also UTF-8, later on this page.

URL

An acronym for Universal Resource Locator, which is a unique identifier (address) of a specific resource, or file, that is available on the World Wide Web (WWW) and other Internet resources. The URL contains the protocol (the method of access) to be used to access the file resource (for example, http:// for World Wide Web pages, ftp:// for file transfers, mailto:// for e-mail, and so forth), the domain name that identifies a specific host computer on the Internet for the file, and the path that specifies the location of the file on that computer.

A URL is a type of URI (Uniform Resource Identifier, formerly called Universal Resource Identifier).

For XML Extensions purposes, a filename specification is considered to be a URL if it begins with “http://”, “https://”, or “file://”.

UTF-8

UTF stands for Unicode Transformation Format. UTF-8 is an encoding scheme (that is, a method of mapping the Unicode code points to a digital representation), which is commonly used under UNIX-style operating systems and in XML documents. Unicode is defined in ISO 10646-1:2000 Annex D and is also described in RFC 2279, as well as section 3.8 of the Unicode 3.0 standard. It is a variable length encoding scheme from 1 to 6 bytes per character. See also Unicode (on page 207).

Valid XML document

See Schema valid XML document (on page 206).

Well-formed XML document

A well-formed XML document is one that conforms to the syntax requirements of XML. A well-formed XML document may or may not be a valid document with respect to a particular XML schema.

XHTML

An acronym for Extensible HyperText Markup Language. When HTML 4.0 is expressed as XML, it is called XHTML. See also HTML (on page 206).

XML

An acronym for Extensible Markup Language. A subset of SGML constituting a particular text markup language for interchange of structured data. The Unicode Standard is the reference character set for XML content. See also Unicode (on page 207).

XML schema

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

XSL

An acronym for Extensible Stylesheet Language. A W3C standard defining XSLT stylesheets for (and in) XML. See also XSLT (on page 208) and W3C (on page 209).

XSLT

An acronym for Extensible Stylesheet Language for Transformations. XSLT is the "Transformations" part of the Extensible Stylesheet Language (XSL). A W3C standard, it is used to transform XML documents to other formats, including HTML, other forms of XML, and plain text. This powerful stylesheet language allows for more complex processing of the XML document's data. See also XSL (on page 208) and W3C (on page 209).

XSLT stylesheet

An XML document that is written in the Extensible Stylesheet Language for Transformations. Note that XSLT stylesheets should not be confused with Cascading Stylesheets (CSS), which are a simple method for adding style, such as fonts, color, and spacing, to a document for final output to a browser; cascading stylesheets are closely related to HTML and XHTML.

W3C

An acronym for World Wide Web Consortium. The main standards body for the World-Wide Web (WWW). W3C works with the global community to establish international standards for client and server protocols that enable online commence and communications on the Internet.

Index

A

All caps, use of as a document convention 3
Allocation of memory 26
Anonymous COBOL data structures 78, 202
Arrays
  empty occurrences 53, 56, 57
  glossary term 205
  sparse 75, 109
ASCII characters 79, 83
Attributes
  COBOL 53, 148
    length 53, 56, 58
    subscript 53, 56, 58, 75, 109
  DISABLE ATTRIBUTES statement, XML 57
  ENABLE ATTRIBUTES statement, XML 58
  unique identifier (uid) 72, 74
  XML 15, 196
  XML DISABLE ATTRIBUTES statement 109
  XML ENABLE ATTRIBUTES statement 109

B

Banner options (slicexsy utility) 195
Batch files, using with example programs 175
Bold type, use of as a document convention 3
Brackets ([ ]), using with
  COBOL syntax 3
  XML Extensions error messages 179

C

Caching
  glossary term 205
  XML documents 26, 53, 58–59, 81
Carriage returns 86
Character encoding 54, 203
  and COBOL 70
  and XML 83
  in UNIX 70
  in Windows 70
  RM_ENCODING environment variable 63, 70
  XML SET ENCODING statement 63, 70, 83
Characters, wide and narrow 79
COBOL
  and XML 12
  attributes 53, 148
  character encoding 54, 70, 203
  considerations
    copy files 8, 20, 75
    data conventions 69
    file management 67
    limitations 78
    optimizations 80
  data structures 11
    anonymous 78, 202
    glossary term 205
  importing from and exporting to XML
      documents 11
  symbol table information 18
cobtoxml utility
  backward compatibility 200
  time stamp checking 203
CodeBridge flags 54, 60, 64, 86
COMPILER-OPTIONS configuration record
  SUPPRESS-XML-SYMBOL-TABLE
      keyword 190, 196
Conventions and symbols used in this manual 3
Copy files
  display status information 77
  listed 8
  statement definitions 20, 75
  terminate application 77
cpy files. See Copy files.

D

Data conventions
  data representation 69
  FILLER data items 71
  intermediate parent names 72
  sparse COBOL records 75
Data items, COBOL. *See also* Data conventions; Data structures, COBOL.
  edited 79
  Internet restrictions 80
  limitations 79
  OCCURS restrictions 80
  size 79
  wide and narrow characters 79
Data naming, in COBOL and XML Extensions 79
Data representation 69
Data structures, COBOL 11
  anonymous 78, 202
  glossary term 205
Data transformations considerations, COBOL and XML 69
Data-names 79
DEPENDING variable 80
Digits, use of in data-names 79
Directory polling 50
  example program 126
Directory search 67–68, 193
Display status information (copy file) 77
Document Object Model, glossary term 206
Document prefix 84
  example program 170
  XML EXPORT FILE statement 28
  XML EXPORT TEXT statement 31
Document type definition (DTD) 40, 84, 202
  glossary term 205
Documentation overview 2
DOM. *See* Document Object Model.
DTD. *See* Document type definition (DTD).

E

Edited COBOL data items 79
Edited data items 86
Elements 13
  unique names 73
Encoding. *See* Character encoding.
Enhancements to XML Extensions 1, 199
  version 1 203
  version 12 199
  version 2 202
  version 9 201
Entity names, defining 40, 84, 202
Environment variables
  PATH 23, 71
  RM_ENCODING 63, 70
  RM_ICONV_NAME 71
  RM_MISSING_HASH 28, 31, 34, 36, 49
  RMPATH 24
  RUNPATH 24, 26, 47–49, 67, 193, 201
Error messages 179
Example programs 8, 89
  batch files, using with 175
  development process, typical 18
  export file and import file 90
  export file and import file with directory polling 126
  export file and import file with OCCURS DEPENDING 103
  export file and import file with sparse arrays 109
  export file and import file with XSLT stylesheets 95
  export file with document prefix 170
  export file, test well-formed file, and validate file 134
  export file, test well-formed text, and validate text 140
  export file, transform file, and import file 147
  export text and import text 120
  import file with missing intermediate parent names 162
  well-formed and validate diagnostic messages 154
Extensible HyperText Markup Language (XHTML) 72, 84
  glossary term 208
Extensible Markup Language (XML). *See also* XML.
  glossary term 208
Extensible Stylesheet Language (XSL), glossary term 208
Extensible Stylesheet Language Transformations (XSLT) 11, 85, 205
  error messages 185
  example of 95
  glossary term 208
  parser (libxslt) 7, 9, 25, 202
  validation 40, 41
Extensions, filename 68, 191
  COBOL source program (.cbl) 176
  model files 26, 191, 193
  schema files 191
  template files 191
  URLs 26, 48, 68, 191, 201
  XSLT stylesheet files 191

External attribute, defined 78
EXTERNAL data items 78
External XSLT stylesheet files 85. *See also* XSLT stylesheet files.
  file naming conventions 68
  glossary term 206

F

File management
  automatic search for files 67
  file naming conventions 68, 191
Filenames. *See also* Extensions, filename.
  conventions used in this manual 3
FILLER data items 71, 79, 84, 86, 111, 202
Flags, CodeBridge 54, 60, 64, 86

G

Glossary terms and definitions 205
Gnome project 8, 9, 202. *See also* libxml and libxslt.

H

HTML. *See* Hypertext Markup Language (HTML).
Hypertext Markup Language (HTML)
  glossary term 206
  vs. XML 11
Hyphen (-), using with
  banner options, slicexsy 195
  optional, RM/COBOL compilation and runtime options 4
  RM_ENCODING environment variable 70
  schema options, slicexsy 195
  XML SET ENCODING statement 63

I

iconv library 71
  glossary term 206
Input and output files, file naming conventions 68
Installation 9
  deployment components 9, 10
  development components 8, 10
  on UNIX 10
  on Windows 10
  system requirements 7
InstantSQL 76
Intermediate parent names 72
  example program 162
Internal XSLT stylesheet files 85, 209
  glossary term 206
Internet address 201. *See also* Referencing Model Files; Universal Resource Locator (URL).
  reading and writing XML documents, restrictions 80
  Universal Resource Locator (URL), glossary term 207
Italic type, use of as a document convention 3

J

Justified data items 69, 86, 202

K

Key combinations, document convention for 3

L

Leading spaces 69, 71, 86, 202
Length attribute 53, 56, 58
libxml 8, 9, 25, 202
libxslt 8, 9, 25, 202
Line feeds 86
Linkage Section 78, 202
lixmlall.cpy 8, 20, 75, 76
lixmldef.cpy 8, 75
lixmldsp.cpy 8, 20, 77
lixmlrpl.cpy 8, 75, 76
lixmltrm.cpy 8, 20, 77
Local character encoding. *See* Character encoding.
Locating files 67
  with envionment variables 23

M

Memory management 26
Messages 179
Model files
  described 17
  file naming conventions 26, 191
  glossary term 206
  referencing 26, 47, 198
  schema 191, 197
  template 191, 196, 203
  types of 196
    internal XSLT stylesheet (.xsl) 191, 197
    schema (.xsd) 196–197
    template (.xtl) 191, 196
  XML EXPORT FILE statement 28
  XML EXPORT TEXT statement 31
  XML IMPORT FILE statement 34, 49
  XML IMPORT TEXT statement 36
MSXML parser 7, 9, 25, 84, 87, 202

N

nonnumeric data items, COBOL 69
numeric data items, COBOL 69

O

Occurrences
  empty 81
  limiting 80
OCCURS DEPENDING clause 103
OCCURS restrictions 80
Online services 4
Organization of this manual 2
Output and input files, file naming conventions 68

P

Parent names. *See* Intermediate parent names.
Parsers, XML 7, 9, 84, 87, 202
PATH environment variable 23, 71

R

Referencing model files 26, 47, 198
Registration, product 4
Related publications 3
REPLACE statement 76
RESOLVE-LEADING-NAME keyword, RUN-FILES-ATTR record 67
RESOLVE-SUBSEQUENT-NAMES keyword, RUN-FILES-ATTR record 67
RM/InfoExpress 67
RM_ENCODING environment variable 63, 70
RM_ICONV_NAME environment variable 71
RM_MISSING_HASH environment variable 28, 31, 34, 36, 49
RMPATH environment variable 24
RUN-FILES-ATTR configuration record
  RESOLVE-LEADING-NAME keyword 67
  RESOLVE-SUBSEQUENT-NAMES keyword 67
RUNPATH environment variable 24, 26, 47–49, 67, 193, 201

S

Sample programs 8, 177
Schema files 87
  filename extension (.xsd) 191
  on UNIX 8
  validation 40, 41, 84, 197, 200
Schema options (slicexsy utility) 195
Schema, valid XML document, glossary term 206
Schema, XML, glossary term 208
SGML (Standardized Generalized Markup Language), glossary term 206
slicexsy utility 200
  and backward compatibility, cobtoxml 200
  command line interface 18, 192
  command line options 195
  described 8, 17
  model files 196, 198
    file naming conventions 26, 191
Spaces 69, 71, 86, 202
Sparse arrays 75, 109
SQL 76
Standardized Generalized Markup Language (SGML), glossary term 206
Statement definitions (copy file) 75
Statements, XML 25
  CLEAR XSL-PARAMETERS 65, 201
  COBOL FILE-NAME 43, 200
  COMPATABILITY MODE 55
  COMPATIBILITY MODE 200
  DISABLE ALL-OCCURRENCES 56, 81
  DISABLE CACHE 58
  ENABLE ALL-OCCURRENCES 57
  ENABLE ATTRIBUTES 58
  ENABLE CACHE 59
  EXPORT FILE 28
  EXPORT TEXT 31
  FIND FILE 51
  FLUSH CACHE 59
  FREE TEXT 44
  GET FLAGS 60, 200
  GET STATUS-TEXT 62
  GET TEXT 45
  GET UNIQUEID 52
  IMPORT FILE 34
  IMPORT TEXT 36
  INITIALIZE 55
  PUT TEXT 46
  REMOVE FILE 46
  RESOLVE DOCUMENT-NAME 47, 200
  RESOLVE MODEL-NAME 49, 200
  RESOLVE SCHEMA-FILE 47, 200
  RESOLVE STYLESHEET-FILE 48, 200
  SET ENCODING 63
  SET FLAGS 64, 86
  SET XSL-PARAMETERS 64, 201
  TERMINATE 56
  TEST WELLFORMED-FILE 38
  TEST WELLFORMED-TEXT 38
  TRACE 60, 201
  TRANSFORM FILE 39
  VALIDATE FILE 40, 197
  VALIDATE FILE 200
  VALIDATE TEXT 41, 197, 200
Status information display (copy file) 62, 77
Structured document 1
  glossary term 207

Stylesheet files
  XML EXPORT FILE statement 28
  XML EXPORT TEXT statement 31
  XML IMPORT FILE statement 34
  XML IMPORT TEXT statement 36
  XML TRANSFORM FILE statement 39
  XML VALIDATE FILE statement 40
  XML VALIDATE TEXT statement 41
Subscript attribute 53, 56, 58, 75, 109
Support services, technical 5
SUPPRESS-XML-SYMBOL-TABLE keyword, COMPILER-OPTIONS configuration record 190, 196
Symbol table information 18
Symbol table, XML-format 199
Symbols and conventions used in this manual 3
System requirements 7

T

Tags, XML 13
Technical support services 5
Template files
  caching 81
  described 196
  filename extension (.xtl) 191, 196
  time stamp checking 203
Terminate application (copy file) 77
Time stamp checking 203
Trailing spaces 69, 71, 86
Truncation 86

U

UNC. *See* Universal Naming Convention.
Underscore (_), using with
  RM_ENCODING environment variable 70
  XML SET ENCODING statement 63
Unicode encoding standard 54, 69, 79, 83, 203
  glossary term 207
Uniform Resource Identifier (URI) 207
Unique element names 72
Unique identifier (uid) 72, 74
Universal Naming Convention (UNC)
  glossary term 207
  locating files 24, 67
  referencing files 198
Universal Resource Locator (URL)
  document type definition (DTD) support 84
  filename extensions 26, 48, 68, 191, 201
  glossary term 207
  reading and writing XML documents, restrictions 80
  recognition of 201
  referencing files 198
  referencing model files 26, 48
  resolving XML files, statements 47–49
UNIX character encoding 70
URI (Uniform Resource Identifier) 207
URL. *See* Universal Resource Locator.
UTF-16 encoding format 83
UTF-8 encoding format 54, 63, 79, 83, 203
  glossary term 208

V

Valid XML document, glossary term 208
Validating schema files 40, 41, 84, 197, 200
Validating XML documents 40, 41, 87, 197
  document type definition (DTD) 84
  document type definitions (DTD) 84
  example programs 134, 140, 154

W

W3C. *See* World Wide Web Consortium (W3C).
Web site services, Liant 4
Well-formed XML document 197
  document type definition (DTD) 84
  example programs 134, 140, 154
  FILLER data items 71
  flattened version 74
  glossary term 208
  schema files 87
  XML statements 27, 38, 40, 41
Whitespace 69, 71, 86
Wide and narrow characters 79
Windows character encoding 70
Working-Storage Section 1, 12, 20, 75
World Wide Web Consortium (W3C) 11
  glossary term 209

X

XHTML. *See* Extensible HyperText Markup Language (XHTML).
XML
  and COBOL 15
  considerations 83
    character encoding 54, 83, 203
    schema files 87
    XSLT stylesheet files 85
  described 11
  glossary term 208
  parsers 7, 9, 84, 87, 202
  schema validation 27, 40, 41, 84
  stylesheet files 28, 31, 34, 36, 39
  symbol table, configuration 190, 196
  symbol table, XML-format 199
  validating 87
  vs. HTML 11
  well-formed XML document 197, 208
  XSLT stylesheet files 15
    internal 191
XML Extensions
  COBOL considerations 67
  data conventions 69
  data file management 67
  data transformation considerations, COBOL and XML 69
  directory search 67–68
  enhancements 1, 199
  error messages 179
  example programs 8, 89
    development process, typical 18
  features, new 201–3
  getting started 17
  handling spaces and whitespace in XML 86
  installation 9
    deployment components 9, 10
    development components 8, 10
    on UNIX 10
    on Windows 10
    system requirements 7
  locating files 23, 67
  model files 196, 198
  overview 11
  sample programs 177
  schema files, validating 200
  statements 200
  symbol table, XML-format 199
  XML considerations 83
XML schema, glossary term 208
xmlif library 9, 24
  described 17
  model files 198
  template files 196
  XSLT stylesheet files 15
    internal 191
XSL. *See* Extensible Stylesheet Language (XSL).
XSLT. *See* Extensible Stylesheet Language Transformations (XSLT); XSLT stylesheet files.
XSLT stylesheet files 85
  caching 53, 81
  example program 95, 100
  external 15, 68
  glossary term 209
  internal 191