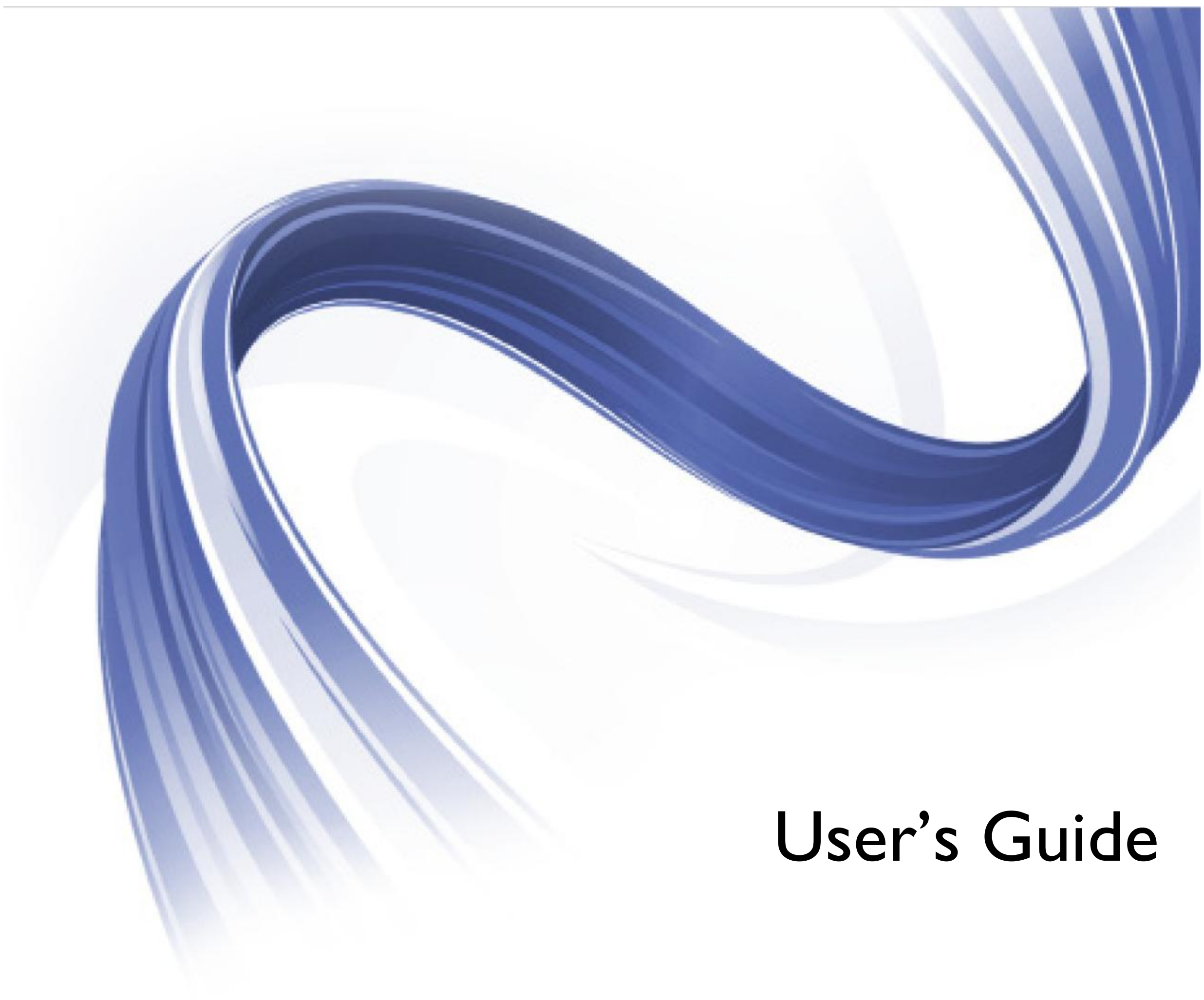
XML Extensions for Enterprise Developer



User's Guide

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

Contents

1. Preface

Welcome to XML Extensions

XML Extensions is Micro Focus Corporation's resource that allows Enterprise Developer 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.

XML Extensions runs on Microsoft Windows operating systems and selected UNIX platforms. Both 32-bit and 64-bit versions are available. It requires Enterprise Developer version 2.2 or later.

Notes

- The term "Windows" in this document refers to Microsoft Windows operating systems, including Microsoft Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, Windows 8, or Windows Server 2012 unless specifically stated otherwise.
- XML Extensions makes use of underlying XML parsers from other vendors. On Windows, development uses 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 technical support to ensure that the update is necessary. On UNIX or Linux systems, XML Extensions links to specific libxml and 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 Micro Focus web site.

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 Enterprise Developer when developing an XML-enabled application.
- **Chapter 5** - XML Considerations provides information specific to using XML when using XML Extensions with Enterprise Developer 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.

This guide also includes a glossary and an index.

Related Publications

For additional information, refer to the following publications:

- Enterprise Developer documentation, available in the product
- 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:

- Words in all capital letters indicate COBOL reserved words, such as statements, phrases, and clauses; acronyms; configuration keywords; environment variables, and COBOL Compiler and Runtime Command line options.
- 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.

- 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.
- 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.
- The symbols found in the COBOL syntax charts are used as follows:
  - Italicized words indicate items for which you substitute a specific value.
  - UPPERCASE WORDS indicate items that you enter exactly as shown (although not necessarily in uppercase).
  - ... indicates indefinite repetition of the last item.
  - | separates alternatives (an either/or choice).
  - [ ] enclose optional items or parameters.
  - { } enclose a set of alternatives, one of which is required.
  - {| |} 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.
- All punctuation must appear exactly as shown.
- 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.
- 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 Microsoft Windows operating systems, including Microsoft Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, Windows 8, or Windows Server 2012, unless specifically stated otherwise.

2. 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 13), 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 Enterprise Developer version 2.2 for Windows. Both 32-bit and 64-bit versions are available.
- 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, visit www.microsoft.com/en-gb/download/default.aspx 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 Enterprise Developer version 2.2 for UNIX. Both 32-bit and 64-bit versions are available.

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, 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 Enterprise Developer compiler**. The XMLGEN compiler directive generates an XML representation of the COBOL program's WORKING-STORAGE data items into a model file used by XML Extensions.
- **Deployment files**. These files are listed in Deployment (on page 13).
- **Copy files** (lixmlall.cpy, lixmldef.cpy, lixmldsp.cpy, lixmlrpl.cpy, and lixmltrm.cpy). For more details, see *Copy Files* (on page 62).
- **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 72). 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 128) for more details. The sample application programs can be found in the XML Extensions sample directory, Samples.

Deployment

The XML Extensions deployment system consists of the following files:

- xmlif COBOL-callable subprogram library (cobxmlext.dll on Windows and cobxmlext.so on UNIX). For more information, see 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 cobmlext.so files and do not need to be installed separately.

Installing XML Extensions

XML Extensions is part of and installed with Enterprise Developer products. After Enterprise Developer is successfully installed XML Extensions is ready for use.

For development, an Enterprise Developer system that includes the Enterprise Developer compiler is required.

For deployment, an Enterprise Developer runtime system, such as Enterprise Server, is required.

Installing on Windows or UNIX

Simply follow the instructions for installing Enterprise Developer onto your system. XML Extensions is installed as part of Enterprise Developer.

Introducing XML Extensions

XML Extensions for Enterprise Developer allows Enterprise Developer 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 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. The Enterprise Developer compiler XMLGEN directive generates an XML-format symbol table into a file called a model 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 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. 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).
```

3. 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 Enterprise Developer compiler, using the XMLGEN compiler directive.
- A COBOL-callable runtime library (cobxmlext.dll on Windows and cobxmlext.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 *XML Extensions Statements Reference (on page 25)*.

Typical Development Process Example

This section provides an example of how to produce an XML-enabled application. These instructions assume that Enterprise Developer including XML Extensions (version 2.2 or later) is 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 72), *Appendix B: XML Extensions Sample Application Programs* (on page 128), and in the XML Extensions samples directory, which is found via the Samples Browser (click on the **Samples** link under your installed Enterprise Developer product).

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

1. *Design the COBOL Data Structure and Program Logic* (on page 20). Develop a COBOL program, or modify an existing one, using XML Extensions statements.
2. *Compile the Program* (on page 20). Use the Enterprise Developer compiler with compiler directive XMLGEN(WS), 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* (on page 21). Test the program and repeat steps 1 and 2, as necessary.
4. *Deploy the Application* (on page 24). Distribute the XML Extensions deployable files. These files consist of the xmlif library and the underlying XML parser that this library uses.

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

Design the COBOL Data Structure 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  Cust-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 the XMLGEN compiler directive of the Enterprise Developer compiler.

The following steps assume you are running in an Enterprise Developer Command Prompt on Windows. UNIX commands are shown at the end of all of the steps.

Compile the program with the following command line:

```
cobol getstarted.cbl xmlgen(ws) noobj;
```

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 62).

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  Cust-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:

```
cobol getstarted.cbl xmlgen(ws) noobj;
run getstarted
```

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

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  Cust-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:

```
cobol getstarted.cbl xmlgen(ws) noobj;
run 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  Cust-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 "Micro Focus" to Cust-Name.
    Move "8310 Capital of Texas Highway, North"
        to Address-1.
    Move "Building 2, Suite 100" to Address-2.
    Move "Austin" to City.
    Move "TX" to State.
    Move 78731 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:

```
cobol getstarted.cbl xmlgen(ws) noobj;
run getstarted
```

This time the XML document, in file Address.xml, is fully populated with data values, as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
 <customer-address xmlns:xtk="http://www.microfocus.com/xcentrisity/
                  \xml-extensions/symbol-table/">
  <cust-name>Micro Focus</cust-name>
  <address-1>8310 Capital of Texas Highway, North</address-1>
  <address-2>Building 2, Suite 100</address-2>
  <address-3>
   <city>Austin</city>
   <state>TX</state>
   <zip>78731</zip>
  </address-3>
 </customer-address>
```

On UNIX, the commands are:

```
cob -i getstarted.cbl -C 'xmlgen(ws)'
cobrun getstarted
```

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 necessary, deploy the model files that you generated. Normally, these files are stored in the same location as the COBOL program files.

How XML Extensions Locates Files

XML Extensions uses the following environment variables to locate various files:

- PATH. The PATH environment variable is used to locate executable programs. An Enterprise Developer Command Prompt window already has PATH set up properly.

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

- *Searching for Files* (on page 26)
- *File Naming Conventions* (on page 56)
- *UNIX Character Encoding* (on page 58)
- *Windows Character Encoding* (on page 58)

4. 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 (cobxmlext.dll) or the shared object on UNIX (cobxmlext.so) that is callable from Enterprise Developer 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 13) and Deployment (on page 13).

XML Extensions statements are grouped into the following categories:

- *Document Processing Statements* (on page 26). These statements are used to process, manipulate, or validate XML documents.
- *Document Management Statements* (on page 38). 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 43). These statements are useful when implementing directory-polling schemes.
- *State Management Statements* (on page 46). 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 XMLGEN compiler directive. 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 44).

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 XMLGEN compiler directive. 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.

XML Extensions uses the CBL_LOCATE_FILE library routine 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 27) and XML EXPORT TEXT (on page 28) 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 30) and XML IMPORT TEXT (on page 32) are available to convert the content of an XML document—either an external file or an internal text string—to a COBOL data item.
- Test and validation statements. XML TEST WELLFORMED-FILE (on page 34), XML TEST WELLFORMED-TEXT (on page 34), XML VALIDATE FILE (on page 35), and XML VALIDATE TEXT (on page 37) are available to verify that an XML document—either an external file or an internal text string—is well-formed or valid.
- Transformation statements. Lastly, XML TRANSFORM FILE (on page 35) and XML TRANSFORM TEXT (on page 35) transform an XML document in an external file or internal text into a new external file or internal text by applying an XSLT stylesheet. The

resulting file or internal text 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* (on page 38), 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). 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 XMLGEN compiler directive that describe the COBOL data item. For more information, see Model Files.
[*StyleSheetName*]	Optional. The name of an external XSLT stylesheet that will be used to transform the generated XML document before it is stored.

Description

The XML EXPORT FILE statement exports the content of the COBOL data item indicated by the *DataItem* parameter. The content of the data item is converted to an XML document using one or more files indicated by the *ModelFileName#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.
```

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.
DocumentLength	The name of a COBOL numeric data item that will contain the length of the generated XML document pointed to by *DocumentPointer* 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 (on page 38), 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). 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 XMLGEN compiler directive that describe the COBOL data item. For more information, see Model Files.
[*StyleSheetName]*	Optional. The name of an external XSLT stylesheet that will be used to transform the generated XML document before it is stored.

Description

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

A block of memory is allocated to hold the generated XML document. The descriptor of this memory block overrides any existing address descriptor in the COBOL pointer data item. The COBOL application is responsible for releasing this memory when it is no longer needed by using XML FREE TEXT (on page 39).

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-DOCUMENT-LENGTH
    "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-DOCUMENT-LENGTH
    "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 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 38), 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).

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 XMLGEN compiler directive that describe the COBOL data item. For more information, see Model Files.

[*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.
DocumentLength	The name of a COBOL numeric data item that contains the length of the XML document pointed to by *DocumentPointer*.
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 38), 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). 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 XMLGEN compiler directive that describe the COBOL data item. For more information, see Model Files.

[*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* and *DocumentLength* parameters. 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-DOCUMENT-LENGTH
    "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-DOCUMENT-LENGTH
    "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 35) and XML VALIDATE TEXT (on page 37) 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.
DocumentLength	The name of a COBOL numeric data item that contains the length of the XML document pointed to by *DocumentPointer*.

Description

The XML TEST WELLFORMED-TEXT statement tests the XML document specified by the *DocumentPointer* and *DocumentLength* parameters 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 35) and XML VALIDATE TEXT (on page 37) 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-POINTER
    MY-DOCUMENT-LENGTH.
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.

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 TRANSFORM TEXT

This statement has the following parameters:

Parameter	Description
InputDocumentPointer	The name of a COBOL pointer data item that points to an XML document that is stored in memory as a text string (the input document).
InputDocumentLength	The name of a COBOL numeric data item that contains the length of the XML document pointed to by *InputDocumentPointer*.
StyleSheetName	The filename of the XSLT stylesheet used for the transformation.
OutputDocumentPointer	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 (the output document).
OutputDocumentLength	The name of a COBOL numeric data item that will contain the length of

the generated XML document pointed to by *OutputDocumentPointer* after successful completion of the statement.

Description

The XML TRANSFORM TEXT statement transforms the XML document specified by the *InputDocumentPointer* and *InputDocumentLength* parameters using the XSLT stylesheet specified by the *StyleSheetName* parameter into a new document specified by the *OutputDocumentPointer* and *OutputDocumentLength* parameters. The new document may or may not be an XML document depending on the XSLT stylesheet.

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 TEXT
    MY-IN-DOCUMENT-POINTER
    MY-IN-DOCUMENT-LENGTH
    "MY-STYLESHEET"
    MY-OUT-DOCUMENT-POINTER
    MY-OUT-DOCUMENT-LENGTH
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 TEXT
    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*.

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. The schema file is 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.
DocumentLength	The name of a COBOL numeric data item that contains the length of the XML document pointed to by *DocumentPointer*.
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*.

Description

The XML VALIDATE TEXT statement tests the XML document specified by the *DocumentPointer* and *DocumentLength* parameters 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. The schema file is 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-POINTER
    MY-DOCUMENT-LENGTH
    "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 38)
- XML FREE TEXT(on page 39)
- XML GET TEXT (on page 39)
- XML PUT TEXT (on page 40)
- XML REMOVE FILE (on page 40)
- 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 41) include the following:
    - XML RESOLVE DOCUMENT-NAME
    - XML RESOLVE MODEL-NAME
    - XML RESOLVE STYLESHEET-FILE
    - XML RESOLVE SCHEMA-FILE

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 (on page 28) statement or the XML GET TEXT statement (described below).

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.
DocumentLength	The name of a COBOL numeric data item that will contain the length of the generated XML document pointed to by *DocumentPointer* 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* and *DocumentLength* parameters.

When the program has finished using the in-memory document, a call to XML FREE TEXT (on page 39) 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-LENGTH
    "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.
DocumentLength	The name of a COBOL numeric data item that contains the length of the XML document pointed to by *DocumentPointer*.
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* and *DocumentLength* parameters 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-LENGTH
    "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.

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 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. XML Extensions uses the CBL_LOCATE_FILE library routine 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.

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.

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* (on page 38), 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).

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 XMLGEN compiler directive that describe the COBOL data item. For more information, see Model Files.

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 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 44)
- XML GET UNIQUEID (on page 45)

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 COBOL applications. Multiple COBOL runtime systems (perhaps running on separate machines on the same local area network) can use directory polling (perhaps to a directory that is available on the network) as a means of improving throughput in processing XML documents.

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 27), 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, it is recommended 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 30).

Before calling XML FIND FILE again (to process the next file), you must call XML REMOVE FILE (on page 40) 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 initialization and termination. Before issuing a call to any other XML Extensions statement, XML INITIALIZE (on page 47) 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 48) 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 27) or XML EXPORT TEXT (on page 28).

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 48). It is possible to force all occurrences to be output by calling XML ENABLE ALL-OCCURRENCES (on page 49).

- COBOL attributes. For each element generated by the statements, XML EXPORT FILE (on page 27) or XML EXPORT TEXT (on page 28), 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 49) prevents attributes from being written (this is the default). Using XML ENABLE ATTRIBUTES (on page 49) 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 50), which sets the default behavior, enables caching of documents. Executing XML DISABLE CACHE (on page 50) disables caching, thus forcing all documents to be loaded each time they are referenced. Executing XML FLUSH CACHE (on page 50) 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 25).

- Flags. The data conversions performed by the statements, XML EXPORT FILE (on page 27), XML EXPORT TEXT (on page 28), XML IMPORT FILE (on page 30), and XML IMPORT TEXT (on page 32), use the library (which is built into the XML Extensions runtime executable) to perform these conversions. By default, the following flags are set: PF_TRAILING_SPACES, PF_LEADING_SPACES, PF_LEADING_MINUS, and PF_ROUNDED.

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

XML GET FLAGS (on page 51) and XML SET FLAGS (on page 53) are available to alter these defaults.

- 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 53) 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 and XML and Character Encoding.
- Tracing. Trace information can be generated to a designated file using the XML TRACE (on page 51) statement.
- Stylesheet parameters. The passing of parameters to stylesheets can be controlled by the XML SET XSL-PARAMETERS (on page 54) and XML CLEAR XSL-PARAMETERS (on page 55) statements.

XML INITIALIZE

This statement has no parameters.

Description

The XML INITIALIZE statement opens a session with XML Extensions. 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 Enterprise Developer runtime system is incompatible, 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 27) and XML EXPORT TEXT (on page 28). 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 49) and XML ENABLE ATTRIBUTES (on page 49). 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 27) and XML EXPORT TEXT (on page 28), 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 48) 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 48) 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.

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.

Note: In managed code the *Flags* argument is restricted to a PIC X(4) COMP-X data item or equivalent. See *Managed Code Considerations* (on page 67) for more information.

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 (on page 44) 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.

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. On UNIX, the local character encoding is specified by the value of the *MF_XMLEXT_LOCAL_ENCODING* environment variable, with a default of MF_LATIN_9 if the variable is not defined.

Note: If the value of the *Encoding* parameter specifies "utf-8", the *MF_XMLEXT_LOCAL_ENCODING* environment variable is ignored. For more information on this environment variable, see *COBOL and Character Encoding* (on page 57).

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 57) for more information.

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.

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.

Note: In managed code the *Flags* argument is restricted to a PIC X(4) COMP-X data item or equivalent. See *Managed Code Considerations* (on page 67) for more information.

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.

Note: In managed code the name/value arguments are restricted to alphanumeric (PIC X) data items or literals. To use numeric, numeric literal (as in the example below), or other data item types use the XML SET XSL-PARAMETERS-NEW statement instead. See *Managed Code Considerations* (on page 67) for more information.

Example

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

XML SET XSL-PARAMETERS-NEW (managed code only)

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-NEW is for *managed code only* and must be paired with the XML END XSL-PARAMETERS-NEW statement. It allows any type of COBOL data items. It 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 (managed code only)

```
XML SET XSL-PARAMETERS-NEW
    "MY-COUNT", 7
XML END XSL-PARAMETERS-NEW.
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.
```

5. COBOL Considerations

This chapter provides information specific to using 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 57)
- *Copy Files* (on page 62)
- Limitations (on page 65)
- *Optimizations* (on page 66)

File Management

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

- *File Naming Conventions* (on page 56)

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 CBL_LOCATE_FILE library routine 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.

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.

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 59)
- *Missing Intermediate Parent Names* (on page 60)
- *Sparse COBOL Records* (on page 62)

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 65), 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 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 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 58).

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 58).

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 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.

MF_XMLEXT_LOCAL_ENCODING Environment Variable

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

Windows Character Encoding

Under Windows, the 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 *MF_XMLEXT_LOCAL_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. 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 COBOL runtime is normally not concerned with the data encoding used by the underlying operating system. However, note that Latin-1 (ISO-8859-1) is important for the U.S. and Latin-9 (ISO-8859-15) is significant for Western Europe because it contains the Euro currency symbol.

The *MF_XMLEXT_LOCAL_ENCODING Environment Variable* (on page 58) may specify the built-in and predefined values of MF_LATIN_1 and MF_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, “MF_LATIN_9”,
“Mf-Latin-9”, and “mflatin9” are equivalent.

If the value of the *MF_XMLEXT_LOCAL_ENCODING* environment variable is not specified, then MF_LATIN_9 is used as the default.

If the value of the *MF_XMLEXT_LOCAL_ENCODING* environment variable is specified with a value that is not MF_LATIN_1 or MF_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 *MF_XMLEXT_LOCAL_ENCODING* environment variable is specific to the iconv library on the platform in use.

Note: Micro Focus does not provide an iconv library. The developer must acquire an appropriate package.

The value of the *MF_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:

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

If the *MF_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 70).

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

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

generates the following well-formed XML sequence:

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

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

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

generates the following syntactically malformed XML sequence:

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

Whenever FILLER data items are present in a data item that is referenced by the XML EXPORT statements, the resulting document is checked to ensure that the resultant XML document is well-formed. 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 60) (see the next topic). Use this technique to determine whether the element name is unique.
- *Unique Identifier* (on page 61). 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  Customer-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" ?>
<customer-address>
  <name>Micro Focus</name>
  <address-1>8310 Capital of Texas Highway North</address-1>
  <address-2>Building 2, Suite 100</address-2>
  <address-3>
    <city>Austin</city>
    <state>TX</state>
    <zip>78731</zip>
  </address-3>
  <time-stamp>13263347</time-stamp>
</customer-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" ?>
<customer-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>
</customer-address>
```

Unique Identifier

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

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

```
<?xml version="1.0" encoding="UTF-8" ?>
  <customer-address uid="V6" line=”18” offset=”0” length=”239”
category=”group” kind="GRP" type=”xsd:string”
xmlns:xtk=”http://www.microfocus.com/xcentricity/xml-
extensions/symbol-table/”>
<name uid=”V7” line=”19” offset=”0” length=”64”
category=”alphanumeric” kind=“ANS” type=”xsd:string”>Wild Hair
Corporation</name>
<address-1 uid=”V8” line=”21” offset=”64” length=”64”
category=”alphanumeric” kind=”ANS” type=”xsd:string”>8911 Hair
Court</address-1>
<address-2 uid=”V9” line=”23” offset=”128” length=”64”
category=”alphanumeric” kind=”ANS” type=”xsd:string”>Sweet
4300</address-2>
<address-3 uid=”V10” line=”24” offset=”192” length=”39”
category=”group” kind=”GRP” type=”xsd:string”>
<city uid=”V11” line=”25” offset=”192” length=”32”
type="alphannumeric" kind="ANS" type=”xsd:string”>Lostin</city>
<state uid=”V12” line=”26” offset=”224” length=”2”
category="alphanumeric" kind="ANS" type=”xsd:string”>TX</state>
<zip uid=”V13” line=”27” offset=”226” length=”5”
category="numeric" kind="NSU" type=”xsd:decimal” usage=”display”
scale=”0” precision=”5”>70707</zip></address-3>
<time-stamp uid=”V14” line=”28” offset=”231” length="8"
category="numeric" kind="NSU" type="xsd:decimal” usage=”display”
scale=”0” precision=”8”>99999999</time-stamp>
  </customer-address>
```

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 or XML IMPORT TEXT statement. This is accomplished by combining the element name and the uid attribute value to form a new element name. For example, <name uid=“V7”>, could be used to generate a new element name “name.V7”.

```
<?xml version="1.0" encoding="UTF-8" ?>
<customer-address>
  <name uid="V7">>Wild Hair Corporation</name>
  <address-1 uid="V8">>8911 Hair Court</address-1>
  <address-2 uid="V9">>Sweet 4300</address-2>
  <city uid="V11">Lostin</city>
  <state uid="V12">TX</state>
  <zip uid="V13">70707</zip>
  <time-stamp uid="V14">99999999</time-stamp>
</customer-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* (on page 62) (as discussed in the following topic)
- *REPLACE Statement Considerations* (on page 63)
- *Displaying Status Information* (on page 63)
- *Application Termination* (on page 64)

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 lixmall.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 arguments passed to a called program or a pointer using the SET statement (for example, into allocated memory). Importing and exporting can also occur with data items having the external attribute. (An *external attribute* is the attribute of a data item obtained by specification of the EXTERNAL clause in the data description entry of the data item or of a data item to which the subject data item is subordinate.)

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 65)
- *Wide and Narrow Characters* (on page 65)
- *Data Item Size* (on page 66)
- *Data Naming* (on page 66)
- *OCCURS Restrictions* (on page 66)
- *Reading, Writing, and the Internet* (on page 66)

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. 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 or XML EXPORT TEXT statements and retrieve data from the data item after using the XML IMPORT FILE or XML IMPORT TEXT statement.

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

- REDEFINES and RENAMES clauses are not allowed.
- FILLER data items must be nonnumeric.
- The data item must be the same size or larger than the data item specified in the model files, but it is not required to be the same data item. For additional information, see Anonymous COBOL Data Structures (on page 64).

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.

Wide and Narrow Characters

XML was developed to use wide (16-bit) Unicode characters as its natural mode. 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 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 *XML Extensions Statements Reference* 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 67).

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.

Empty Occurrences

On output, the XML EXPORT FILE or XML EXPORT TEXT 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 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, XML DISABLE CACHE, and XML FLUSH CACHE. By default, caching is enabled.

Managed Code Considerations

XML Extensions is available in .NET managed code on Windows.

Compilation Options

To compile for .NET managed code 32-bit add the following compiler directives to your command line, directives file, or in the Additional Directives box in your project's properties.

```
ilgen
iltarget(x86)
ilref(MicroFocus.COBOL.XmlExtensions)
ilref(MicroFocus.COBOL.XmlExtensions.Interop)
```

To compile 64-bit change `iltarget(x86)` above to `iltarget(x64)`. To compile a called subprogram change `ilgen` above to `ilgen(sub)`; that will produce a DLL rather than an EXE.

New XML Statement

One new statement has been added for managed code only. It must be terminated by the appropriate END statement as shown below.

```
XML SET XSL-PARAMETERS-NEW
   <name-value-pairs>
XML END XSL-PARAMETERS-NEW.
```

This new statement is documented on page 55. It functions just like the corresponding XML SET XSL-PARAMETERS statement except that it allows any data type for the value arguments in managed code. The XML SET XSL-PARAMETERS statement can be used in managed code but it only allows alphanumeric (PIC X) data items as arguments.

XML Statement Restrictions

The XML GET FLAGS and XML SET FLAGS statements have a restriction in managed code. The Flags argument must be a PIC X(4) COMP-X data item or equivalent [for example, PIC 9(10) BINARY(4) is equivalent]. If you wish to use a different numeric type for Flags in your COBOL program then you can use the MOVE statement to or from the XML-Flags data item that is defined in the XML-data-group in lixmldef.cpy, as shown in the following examples.

```
XML GET FLAGS XML-Flags.
MOVE XML-Flags TO Your-Flags.
```

```
MOVE Your-Flags TO XML-Flags.
XML SET FLAGS XML-Flags.
```

6. XML Considerations

This chapter provides information specific to using XML when using XML Extensions with Enterprise Developer 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 69)
- *XSLT Stylesheet Files* (on page 70)
- *Handling Spaces and Whitespace in XML* (on page 70)
- *Schema Files* (on page 71)

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 57).

Document Type Definition Support

A DTD is 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 specific statements, XML TRANSFORM FILE and XML TRANSFORM TEXT, which are used for performing XSLT stylesheet transformations. In addition, the import and export statements, XML IMPORT FILE, XML IMPORT TEXT, XML EXPORT FILE, and XML EXPORT TEXT, 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 or edited data items.

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.

Notes:

- 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 69).
- On UNIX systems, the underlying XML parser, libxml, does not support schema validation.

7. 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 128), 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* (on page 72)
- *Example 2: Export File and Import File with XSLT Stylesheets* (on page 77)
- *Example 3: Export File and Import File with OCCURS DEPENDING* (on page 82)
- *Example 4: Export File and Import File with Sparse Arrays* (on page 86)
- *Example 5: Export Text and Import Text* (on page 94)
- *Example 6: Export File and Import File with Directory Polling* (on page 98)
- *Example 7: Export File, Test Well-Formed File, and Validate File* (on page 104)
- *Example 8: Export Text, Test Well-Formed Text, and Validate Text* (on page 108)
- *Example 9: Export File, Transform File, and Import File* (on page 112)
- *Example 10: Diagnostic Messages* (on page 117)
- *Example 11: Import File with Missing Intermediate Parent Names* (on page 121)

Additionally, three batch files are provided to facilitate use of the example programs. See *Example Batch Files* (on page 126).

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

Development for Example 1

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 1

The following DOS commands may be entered into a batch file. These commands build and execute example01.int.

Line	Statement
1	cobol example01.cbl xmlgen(ws) noobj;
2	start /w run example01

Line 1 compiles the example01.cbl source file and generates an XML-format model file named **example01.xml**.

Line 2 executes example01.int. 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 Address-Struct (as defined in the copy file, s-struct.cpy) to an XML document with the filename of address01.xml using the XML EXPORT FILE statement.

Next, the content of the XML document is imported from the file, address01.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, s-struct.cpy, is as follows:

```
*
* Title: s-struct.cpy: XML Extensions sample data structure.
*
* Copyright © Micro Focus 2008-2013.
*     All rights reserved.
*
* Version Identification:
*   $Revision$
*   $Date$
*   $URL$
*
 01  Address-Struct.
```

```
      02  Name           Pic X(64)
              Value "Specialty Cowboy Boots Company".
      02  Address-1      Pic X(64)
              Value "1050 North San Antonio Street".
      02  Address-2      Pic X(64) Value "Suite 200".
      02  Address-3.
          03  City       Pic X(32) Value "Austin".
          03  State      Pic X(2) Value "TX".
          03  Zip        Pic 9(5) Value 78701.
      02  Country        Pic X(64)
              Value "United States of America".
      02  Time-Stamp     Pic X(23).
```

This data item stores company address information. 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) COMP-5.
        88  XML-IsSuccess      VALUE XML-Success.
        88  XML-OK             VALUE XML-WarningLimit
                               THROUGH XML-StatusNonFatal.
        88  XML-IsDirectoryEmpty
                               VALUE  XML-InformDirectoryEmpty.
    03  XML-Status-Edited      PIC +9(4).
    03  XML-StatusText         PIC X(80).
    03  XML-MoreFlag           PIC X COMP-X VALUE 0.
        88 XML-NoMore          VALUE 0.
    03  XML-UniqueID           PIC X(40).
    03  XML-Flags              PIC X(4) COMP-X.
    03  XML-COBOL-Version      PIC X(4) COMP-X VALUE 12. *>Used by
                                                      XMLSetVersion
    03  XML-XMLIF-Version      PIC X(4) COMP-X 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, example01.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
Move … To Time-Stamp.	Populate the Time-Stamp field.

`XML EXPORT FILE`
`    Address-Struct`
`    "address01.xml"`
`    "Address-Struct".`
	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
`Initialize Address-Struct.`	Ensure that the Address-Struct item contains no data.

`XML IMPORT FILE`
`    Address-Struct`
`    "address01.xml"`
`    "Address-Struct".`
	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.

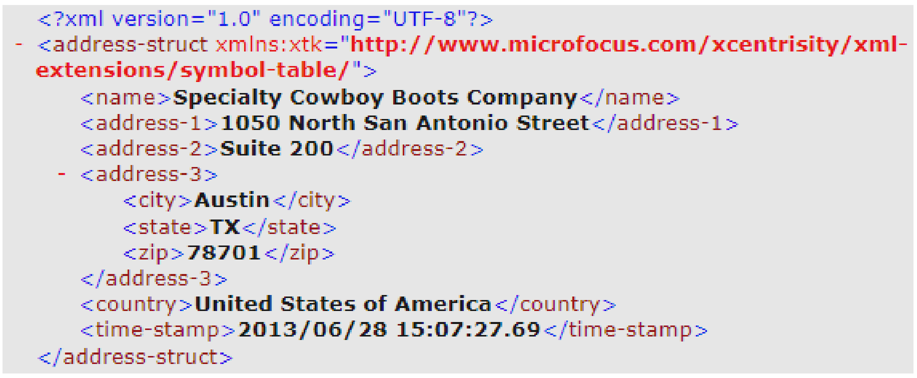
Running the program (run example01) produces the following display:

```
Example-01 - Illustrate EXPORT FILE & IMPORT FILE

address01.xml exported by XML EXPORT FILE

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                         TX78701
Country:      United States of America
Time-Stamp:   2013/06/28  15:07:27.69

address01.xml imported by XML IMPORT FILE

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                         TX78701
Country:      United States of America
Time-Stamp:   2013/06/28  15:07:27.69

You may inspect 'address01.xml'
```

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

XML Document (Example 1)

Microsoft Internet Explorer may be used to view the generated XML document, address01.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"?>
- <address-struct xmlns:xtk="http://www.microfocus.com/xcentrisity/xml-
  extensions/symbol-table/">
      <name>Specialty Cowboy Boots Company</name>
      <address-1>1050 North San Antonio Street</address-1>
      <address-2>Suite 200</address-2>
    - <address-3>
          <city>Austin</city>
          <state>TX</state>
          <zip>78701</zip>
      </address-3>
      <country>United States of America</country>
      <time-stamp>2013/06/28 15:07:27.69</time-stamp>
  </address-struct>
```

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 72). 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.

This example uses the following XML statements:

- XML INITIALIZE, which initializes or opens a session with XML Extensions.
- XML EXPORT FILE, which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE, which reads an XML document (from a file) into a COBOL data item.
- XML TERMINATE, 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 Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 2

Use the same commands as for example 1 except substitute example02 for example01.

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 Address-Struct to an XML document with the filename of address02.xml using the XML EXPORT FILE statement.

Next, the content of the XML document is imported from the file, address02.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 is the same as example 1.

This data item stores company address information. 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. All of this is the same as example 1.

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, example02.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**
`Move … To Time-Stamp.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`    Address-Struct`
`    "address02.xml"`
`    "Address-Struct"`
`    "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**
`Initialize Address-Struct.`	Ensure that the Address-Struct structure contains no data.

`XML IMPORT FILE`
`    Address-Struct`
`    "address02.xml"`
`    "Address-Struct"`
`    "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)

Same as in example 1.

Status Display Logic (Example 2)

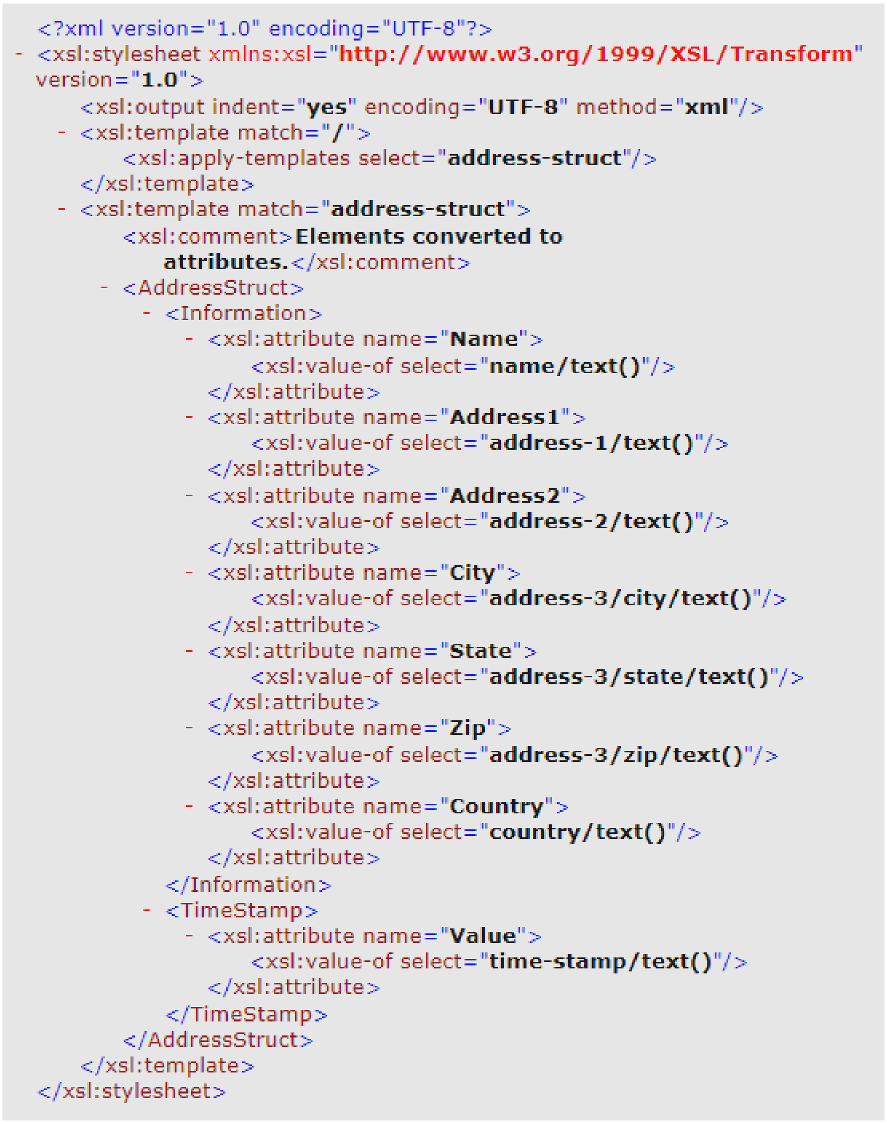
Same as in example 1.

XSLT Stylesheets for Example 2

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

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

toext.xsl (Example 2)



```
  <?xml version="1.0" encoding="UTF-8"?>
- <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
     <xsl:output indent="yes" encoding="UTF-8" method="xml"/>
   - <xsl:template match="/">
        <xsl:apply-templates select="address-struct"/>
     </xsl:template>
   - <xsl:template match="address-struct">
        <xsl:comment>Elements converted to
           attributes.</xsl:comment>
      - <AddressStruct>
         - <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>
            - <xsl:attribute name="Country">
                 <xsl:value-of select="country/text()"/>
              </xsl:attribute>
           </Information>
         - <TimeStamp>
            - <xsl:attribute name="Value">
                 <xsl:value-of select="time-stamp/text()"/>
              </xsl:attribute>
           </TimeStamp>
        </AddressStruct>
     </xsl:template>
  </xsl:stylesheet>
```

toint.xsl (Example 2)

```
  <?xml version="1.0" encoding="UTF-8"?>
- <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
    <xsl:output encoding="UTF-8" method="xml"/>
  - <xsl:template match="/">
      <xsl:apply-templates select="AddressStruct"/>
    </xsl:template>
  - <xsl:template match="AddressStruct">
      <xsl:comment>Attributes converted back to
        elements.</xsl:comment>
    - <address-struct>
      - <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>
      - <country>
          <xsl:value-of select="Information/@Country"/>
        </country>
      - <time-stamp>
          <xsl:value-of select="TimeStamp/@Value"/>
        </time-stamp>
      </address-struct>
    </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.

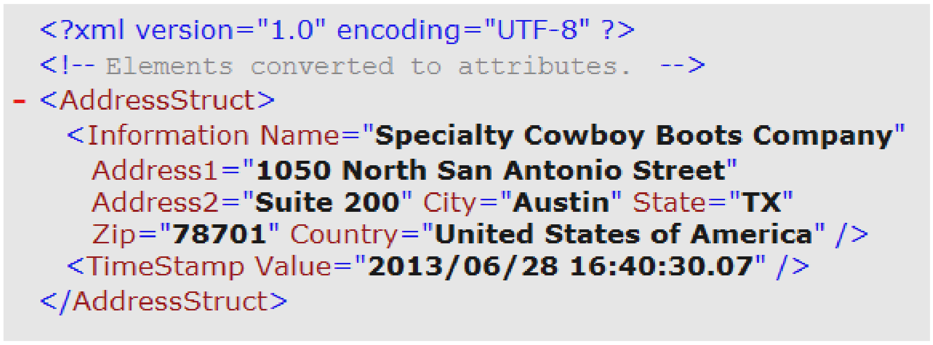
Running the program (run example02) produces the following display:

```
Example-02 - Illustrate EXPORT FILE & IMPORT FILE with XSLT
stylesheets

address02.xml exported by XML EXPORT FILE

Name:        Specialty Cowboy Boots Company
Address-1:   1050 North San Antonio Street
Address-2:   Suite 200
Address-3:   Austin                        TX78701
Country:     United States of America
Time-Stamp:  2013/06/28  16:40:30.07
```

```
address02.xml imported by XML IMPORT FILE

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                          TX78701
Country:      United States of America
Time-Stamp:   2013/06/28  16:40:30.07

You may inspect 'address02.xml'

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

XML Document (Example 2)

Microsoft Internet Explorer may be used to view the generated XML document, address02.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" ?>
  <!-- Elements converted to attributes.  -->
- <AddressStruct>
    <Information Name="Specialty Cowboy Boots Company"
      Address1="1050 North San Antonio Street"
      Address2="Suite 200" City="Austin" State="TX"
      Zip="78701" Country="United States of America" />
    <TimeStamp Value="2013/06/28 16:40:30.07" />
  </AddressStruct>
```

This XML document differs from the document generated in *Example 1: Export File and Import File* (on page 72). 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 72). However, the data item has been modified so that an OCCURS DEPENDING clause is present.

This example uses the following XML statements:

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

Development for Example 3

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 3

Use the same commands as for example 1 except substitute example03 for example01.

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 Address-Struct3 (as defined in the copy file, s-struct3.cpy) to an XML document with the filename of address03.xml using the XML EXPORT FILE statement.

Next, the content of the XML document is imported from the file, address03.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, s-struct3.cpy, is as follows:

```
      *
      * Title: s-struct3.cpy: XML Extensions sample data
structure.
      *
      * Copyright © Micro Focus 2008-2013.
      *     All rights reserved.
      *
      * Version Identification:
      *   $Revision$
      *   $Date$
      *   $URL$
      *
       01  Address-Struct3.
           02  Time-Stamp    Pic X(23).
           02  Name          Pic X(64)
                   Value "Cowboy Hats and More Company".
           02  City          Pic X(32) Value "San Marcos".
           02  State         Pic X(2) Value "TX".
           02  Zip           Pic 9(5) Value 78666.
           02  Country       PIC X(64)
                   Value "United States of America".
           02  Address-Lines Pic 9.
           02  Address-Line  Pic X(64)
                   Occurs 1 to 5 times Depending on Address-Lines.
       01 Struct-Name       Pic X(15) Value "Address-Struct3".
       01 Street            PIC X(64) Value
                                "504 Broadway Street".
       01 Building          PIC X(64) Value "Building 1".
       01 Suite             PIC X(64) Value "Suite 142".
```

This data item stores company address information. 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. All of this is the same as examples 1 and 2.

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, example03.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**
`Move … To Time-Stamp.`	Populate the Time-Stamp field.

`Move 3 to Address-Lines.`
`Move … to Address-Line(1).`
`Move … to Address-Line(2).`
`Move … to Address-Line(3).`
	Ensure that Address Lines contain proper information.

`XML EXPORT FILE`
`    Address-Struct3`
`    "address03.xml"`
`    "Address-Struct3".`
	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**
`Initialize Address-Struct3.`	Ensure that the address structure contains no data.

`XML IMPORT FILE`
`    Address-Struct3`
`    "address03.xml"`
`    "Address-Struct3".`
	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)

Same as example 1.

Termination Test Logic (Example 3)

Same as example 1.

Status Display Logic (Example 3)

Same as example 1.

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)

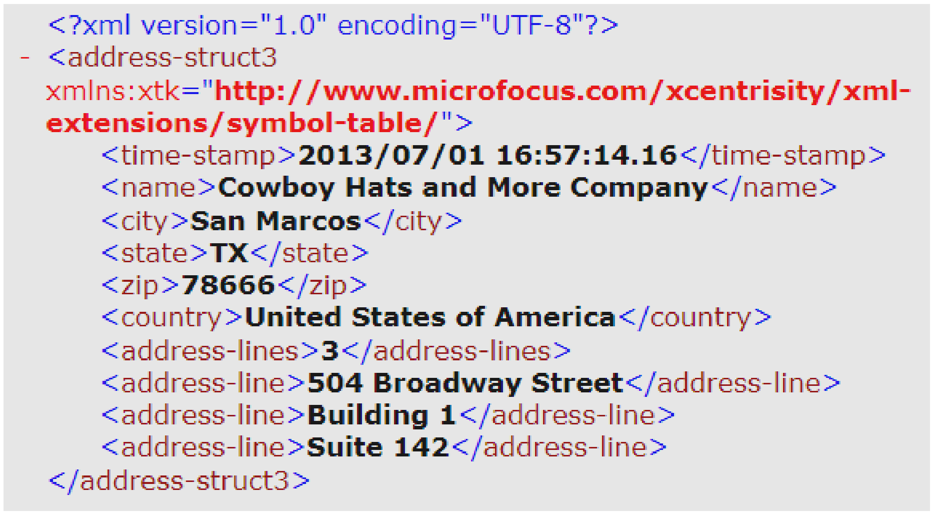
Note: Pressing a key will terminate the program.

Running the program produces the following display:

```
Example-03 - Illustrate EXPORT FILE & IMPORT FILE with OCCURS
DEPENDING

address03.xml exported by XML EXPORT FILE

Name:              Cowboy Hats and More Company
Address-Line(1):   504 Broadway Street
Address-Line(2):   Building 1
Address-Line(3):   Suite 142
City State Zip:    San Marcos                        TX78666
Country:           United States of America
Time-Stamp:        2013/07/01  16:57:14.16

address03.xml imported by XML IMPORT FILE

Name:              Cowboy Hats and More Company
Address-Line(1):   504 Broadway Street
Address-Line(2):   Building 1
Address-Line(3):   Suite 142
City State Zip:    San Marcos                        TX78666
Country:           United States of America
Time-Stamp:        2013/07/01  16:57:14.16

You may inspect 'address03.xml'

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

XML Document (Example 3)

Microsoft Internet Explorer may be used to view the generated XML document, address03.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"?>
- <address-struct3
  xmlns:xtk="http://www.microfocus.com/xcentrisity/xml-
  extensions/symbol-table/">
      <time-stamp>2013/07/01 16:57:14.16</time-stamp>
      <name>Cowboy Hats and More Company</name>
      <city>San Marcos</city>
      <state>TX</state>
      <zip>78666</zip>
      <country>United States of America</country>
      <address-lines>3</address-lines>
      <address-line>504 Broadway Street</address-line>
      <address-line>Building 1</address-line>
      <address-line>Suite 142</address-line>
  </address-struct3>
```

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, which initializes or opens a session with XML Extensions.
- XML EXPORT FILE, which constructs an XML document (as a file) from the content of a COBOL data item.
- XML IMPORT FILE, which reads an XML document (from a file) into a COBOL data item.

- XML ENABLE ATTRIBUTES, 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, 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, which causes all occurrences of a data item to be exported to an XML document.
- XML DISABLE ALL-OCCURRENCES, 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, which terminates or closes the session with XML Extensions.

Development for Example 4

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 4

Use the same commands as for example 1 except substitute example04 for example01.

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 to several XML documents with the filenames of table1.xml, table2.xml, table3.xml, and table4.xml using the XML EXPORT FILE statement. All four 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 89).

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 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. All of this is the same as in previous examples.

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, example04.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 DISABLE ATTRIBUTES If Not XML-OK Go To Z. XML DISABLE ALL-OCCURRENCES If Not XML-OK Go To Z.	Selectively ENABLE or DISABLE ATTRIBUTES and ALL-OCCURRENCES. All four combinations of ENABLE and DISABLE are used for tables 1 – 4.
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)

Same as example 1.

Termination Test Logic (Example 4)

Same as example 1.

Status Display Logic (Example 4)

Same as example 1.

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 produces the following display:

```
Example-04 - Illustrate EXPORT FILE & 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.)

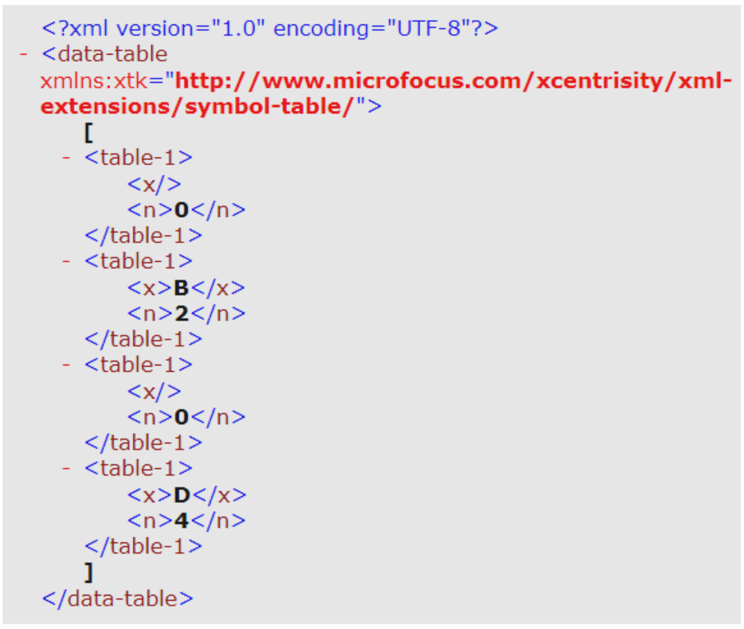
The files table1.xml, table2.xml, table3.xml, and table4.xml are generated with XML EXPORT FILE statements. All of these documents were generated from the same COBOL content. The files, table5.xml and table6.xml, which are supplied with the example, describe the same COBOL content.

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.

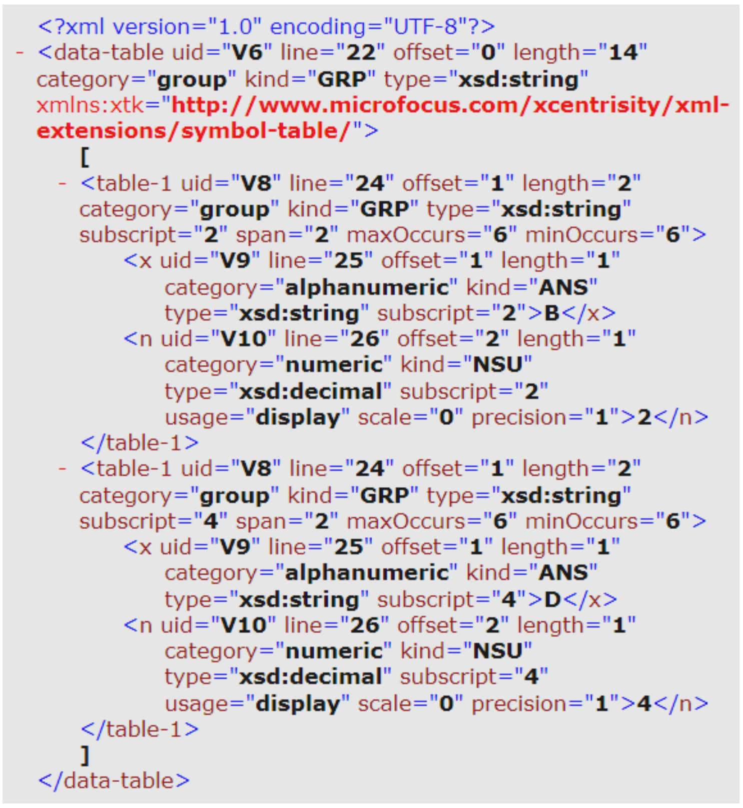
This example also uses FILLER data items. The left brace ([) and right brace (]) characters were defined within the data item as FILLER. The text associated with the FILLER is placed in the XML document without any tags.



```
  <?xml version="1.0" encoding="UTF-8"?>
- <data-table
  xmlns:xtk="http://www.microfocus.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)

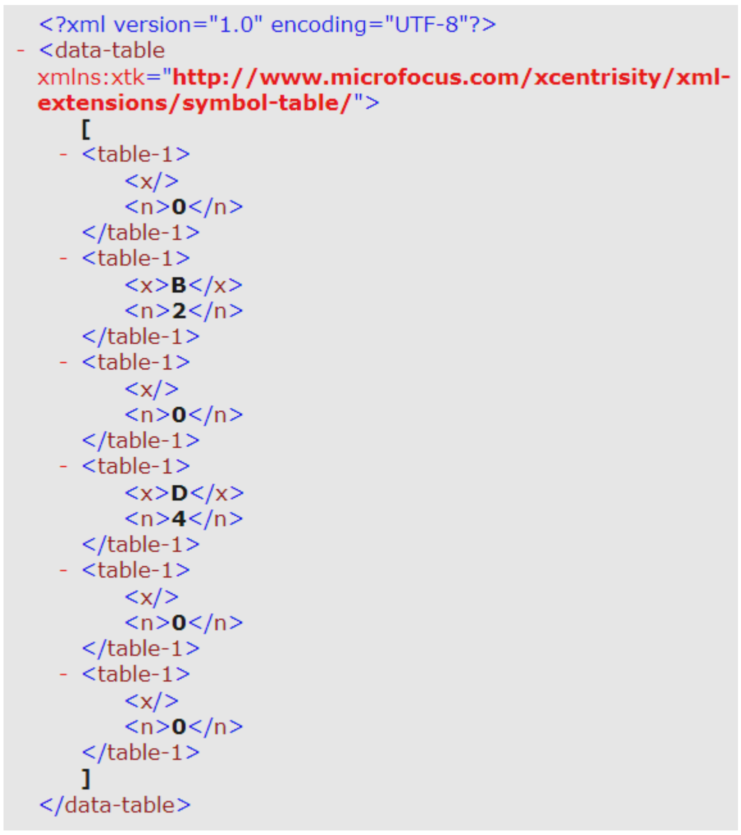
The XML ENABLE ATTRIBUTES and XML DISABLE ALL-OCCURRENCES statements are used to determine the content of this file. Since each non-empty occurrence now contains a subscript attribute, none of the empty occurrences are generated.



```
  <?xml version="1.0" encoding="UTF-8"?>
- <data-table uid="V6" line="22" offset="0" length="14"
  category="group" kind="GRP" type="xsd:string"
  xmlns:xtk="http://www.microfocus.com/xcentrisity/xml-
  extensions/symbol-table/">
      [
    - <table-1 uid="V8" line="24" offset="1" length="2"
      category="group" kind="GRP" type="xsd:string"
      subscript="2" span="2" maxOccurs="6" minOccurs="6">
          <x uid="V9" line="25" offset="1" length="1"
              category="alphanumeric" kind="ANS"
              type="xsd:string" subscript="2">B</x>
          <n uid="V10" line="26" offset="2" length="1"
              category="numeric" kind="NSU"
              type="xsd:decimal" subscript="2"
              usage="display" scale="0" precision="1">2</n>
      </table-1>
    - <table-1 uid="V8" line="24" offset="1" length="2"
      category="group" kind="GRP" type="xsd:string"
      subscript="4" span="2" maxOccurs="6" minOccurs="6">
          <x uid="V9" line="25" offset="1" length="1"
              category="alphanumeric" kind="ANS"
              type="xsd:string" subscript="4">D</x>
          <n uid="V10" line="26" offset="2" length="1"
              category="numeric" kind="NSU"
              type="xsd:decimal" subscript="4"
              usage="display" scale="0" precision="1">4</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://www.microfocus.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://www.microfocus.com/xcentrisity/xml-
    extensions/symbol-table/" type="xsd:string" kind="GRP"
    category="group" length="14" offset="0" line="22" uid="V6">
    [
  - <table-1 type="xsd:string" kind="GRP" category="group" length="2"
      offset="1" minOccurs="6" maxOccurs="6" span="2" subscript="1"
      line="24" uid="V8">
      <x type="xsd:string" kind="ANS" category="alphanumeric"
        length="1" offset="1" subscript="1" line="25" uid="V9" />
      <n type="xsd:decimal" kind="NSU" category="numeric"
        length="1" offset="2" precision="1" scale="0" usage="display"
        subscript="1" line="26" uid="V10">0</n>
    </table-1>
  - <table-1 type="xsd:string" kind="GRP" category="group" length="2"
      offset="1" minOccurs="6" maxOccurs="6" span="2" subscript="2"
      line="24" uid="V8">
      <x type="xsd:string" kind="ANS" category="alphanumeric"
        length="1" offset="1" subscript="2" line="25" uid="V9">B</x>
      <n type="xsd:decimal" kind="NSU" category="numeric"
        length="1" offset="2" precision="1" scale="0" usage="display"
        subscript="2" line="26" uid="V10">2</n>
    </table-1>
  - <table-1 type="xsd:string" kind="GRP" category="group" length="2"
      offset="1" minOccurs="6" maxOccurs="6" span="2" subscript="3"
      line="24" uid="V8">
      <x type="xsd:string" kind="ANS" category="alphanumeric"
        length="1" offset="1" subscript="3" line="25" uid="V9" />
      <n type="xsd:decimal" kind="NSU" category="numeric"
        length="1" offset="2" precision="1" scale="0" usage="display"
        subscript="3" line="26" uid="V10">0</n>
    </table-1>
  - <table-1 type="xsd:string" kind="GRP" category="group" length="2"
      offset="1" minOccurs="6" maxOccurs="6" span="2" subscript="4"
      line="24" uid="V8">
      <x type="xsd:string" kind="ANS" category="alphanumeric"
        length="1" offset="1" subscript="4" line="25" uid="V9">D</x>
      <n type="xsd:decimal" kind="NSU" category="numeric"
        length="1" offset="2" precision="1" scale="0" usage="display"
        subscript="4" line="26" uid="V10">4</n>
    </table-1>
  - <table-1 type="xsd:string" kind="GRP" category="group" length="2"
      offset="1" minOccurs="6" maxOccurs="6" span="2" subscript="5"
      line="24" uid="V8">
      <x type="xsd:string" kind="ANS" category="alphanumeric"
        length="1" offset="1" subscript="5" line="25" uid="V9" />
      <n type="xsd:decimal" kind="NSU" category="numeric"
        length="1" offset="2" precision="1" scale="0" usage="display"
        subscript="5" line="26" uid="V10">0</n>
    </table-1>
  - <table-1 type="xsd:string" kind="GRP" category="group" length="2"
      offset="1" minOccurs="6" maxOccurs="6" span="2" subscript="6"
      line="24" uid="V8">
      <x type="xsd:string" kind="ANS" category="alphanumeric"
        length="1" offset="1" subscript="6" line="25" uid="V9" />
      <n type="xsd:decimal" kind="NSU" category="numeric"
        length="1" offset="2" precision="1" scale="0" usage="display"
        subscript="6" line="26" uid="V10">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, which initializes or opens a session with XML Extensions.
- XML EXPORT TEXT, which constructs an XML document (as a text string) from the content of a COBOL data item.
- XML IMPORT TEXT, which reads an XML document (from a text string) into a COBOL data item.
- XML PUT TEXT, which copies an XML document from a text string to a data file.
- XML FREE TEXT, which releases the memory that was allocated by XML EXPORT TEXT to hold the XML document as a text string.
- XML TERMINATE, which terminates or closes the session with XML Extensions.

Development for Example 5

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 5

Use the same commands as for example 1 except substitute example05 for example01.

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 72), 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 Address-Struct (as defined in the copy file, s-struct.cpy) to an in-memory XML document as defined by the variables, *Document-Pointer* and *Document-Length*, using the XML EXPORT TEXT statement.

Next, the content of the XML document is imported from the in-memory text string, 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, Address-Struct, is the same as in example 1.

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. All of this is the same as in previous examples.

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, example05.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
`Move … Time-Stamp.`	Populate the Time-Stamp field.

`XML EXPORT TEXT`
`    Address-Struct`
`    Document-Pointer`
`    Document-Length`
`    "Address-Struct".`
	Execute the XML EXPORT TEXT statement specifying:
	the data item address,
	the XML document text pointer,
	the XML document text length,
	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
`Initialize Address-Struct.`	Ensure that the address structure contains no data.

`XML IMPORT TEXT`
`    Address-Struct`
`    Document-Pointer`
`    Document-Length`
`    "Address-Struct".`
	Execute the XML IMPORT TEXT statement specifying:
	the data item address,
	the XML document text pointer,
	the XML document text length,
	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`
`    Document-Length`
`    "address05".`
	Execute the XML PUT TEXT statement specifying:
	the XML document text pointer
	the XML document text length,
	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)

Same as example 1.

Termination Test Logic (Example 5)

Same as example 1.

Status Display Logic (Example 5)

Same as example 1.

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)

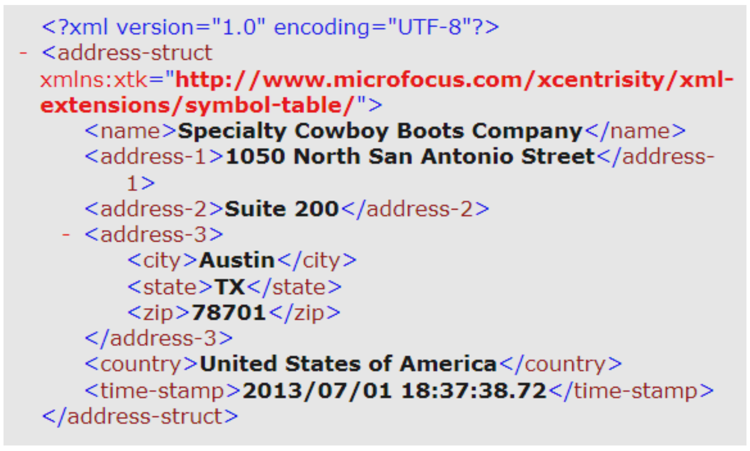
Note: Pressing a key will terminate the program.

Running the program (run example05) produces the following display:

```
Example-05 - Illustrate EXPORT TEXT & IMPORT TEXT

Document exported by XML EXPORT TEXT

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                          TX78701
Country:      United States of America
Time-Stamp:   2013/07/01  18:37:38.72

Document imported by XML IMPORT TEXT

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                          TX78701
Country:      United States of America
Time-Stamp:   2013/07/01  18:37:38.72

Document memory written by XML PUT TEXT
Document memory released by XML FREE TEXT

You may inspect 'address05.xml'

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

XML Document (Example 5)

Microsoft Internet Explorer may be used to view the generated XML document, address05.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"?>
- <address-struct
  xmlns:xtk="http://www.microfocus.com/xcentrisity/xml-
  extensions/symbol-table/">
     <name>Specialty Cowboy Boots Company</name>
     <address-1>1050 North San Antonio Street</address-
        1>
     <address-2>Suite 200</address-2>
   - <address-3>
        <city>Austin</city>
        <state>TX</state>
        <zip>78701</zip>
     </address-3>
     <country>United States of America</country>
     <time-stamp>2013/07/01 18:37:38.72</time-stamp>
  </address-struct>
```

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 43).

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

Development for Example 6

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 6

The following commands build and execute example06.int.

Line	Statement
1	cobol example06.cbl xmlgen(ws) noobj;
2	mkdir stamp
3	start /w run example06

This is very similar to example 1 except that mkdir is used to create the stamp subdirectory used by this example prior to executing the program.

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. All of this is the same as in previous examples.

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, example06.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
`Perform 5 Times`	Iterate to export 5 files.

`  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.

`  Move … To Time-Stamp.`	Populate the Time-Stamp field.

`  XML EXPORT FILE`
`      Time-Stamp`
`      Unique-File-Name`
`      "Time-Stamp".`
	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.
`End-Perform.`	

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`
`XML FIND FILE`
`"stamp"`
`Unique-File-Name`
`If XML-IsSuccess`
`Exit Perform`
`End-If`
`If XML-IsDirectoryEmpty`
`Perform Compute-Curr-`
`Time`
`If Curr-Time > Stop-`
`Time`
`Exit Perform`
`End-If`
`Call`
`"CBL_THREAD_SLEEP" …`
`End-If`
`If Not XML-OK`
`Go To Z`
`End-If`
`End-Perform`
	Inner perform loop. Iterate until Exit Perform.
	Execute XML FIND FILE parameters:
	directory name
	and filename.
	If the statement returned success,
	exit the paragraph.
	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 the statement terminates unsuccessfully,
	go to the termination logic.
	The end of the inner perform loop.

`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`
`"Time-Stamp"`
`If Not XML-OK Go To Z`
`End-If`
	Import the file that was found using:
	the data item,
	the filename,
	and the *ModelFileName#DataFileName* parameter value.
	If the statement terminates unsuccessfully, go to the termination logic.

`XML REMOVE FILE`
`Unique-File-Name`
`If Not XML-OK Go To Z`
`End-If`
	Remove the file that has just been processed; otherwise, find it again.
	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 (run example06) 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-06 - Illustrate EXPORT FILE & IMPORT FILE with directory
polling
stamp/a{a258c50d-a15e-493b-a29d-cc0e782b5f54}.xml exported by
XMLExport
Contents: 2013/07/01  19:22:39.07
stamp/a{9318803d-1b46-486c-a59b-f7dcc92c4d2f}.xml exported by
XMLExport
Contents: 2013/07/01  19:22:39.07
stamp/a{3a2b7d60-6065-4785-bdf3-ab388992079d}.xml exported by
XMLExport
Contents: 2013/07/01  19:22:39.07
stamp/a{4a4e8482-d3a4-492e-a79d-ec6d967cd4e6}.xml exported by
XMLExport
Contents: 2013/07/01  19:22:39.07
stamp/a{30cd08ac-0edf-4885-a106-4acdcb8caada}.xml exported by
XMLExport
Contents: 2013/07/01  19:22:39.07

You may display the 'stamp' directory

Press a key to continue:
```

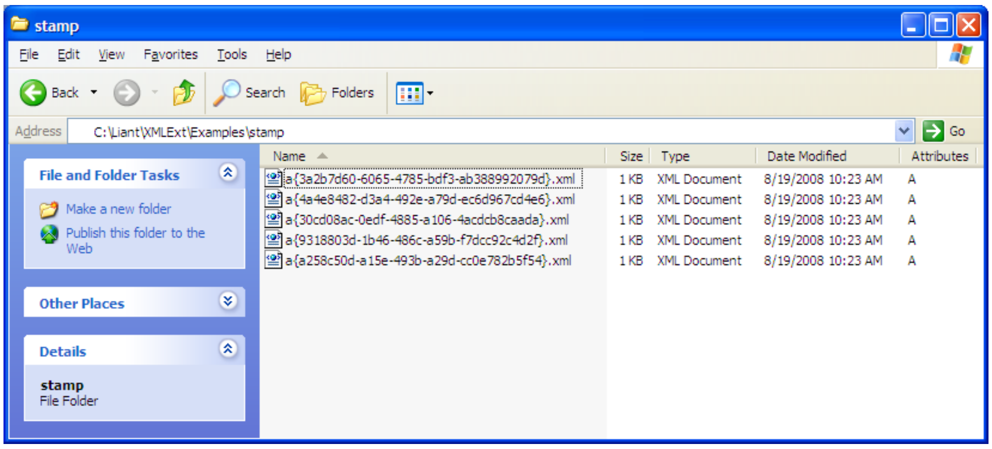
Second Display

Note: Pressing a key will terminate the program.

```
 stamp\a{30cd08ac-0edf-4885-a106-4acdcb8caada}.xml imported by
XMLImport
Contents: 2013/07/01  19:22:39.07
 stamp\a{3a2b7d60-6065-4785-bdf3-ab388992079d}.xml imported by
XMLImport
Contents: 2013/07/01  19:22:39.07
 stamp\a{4a4e8482-d3a4-492e-a79d-ec6d967cd4e6}.xml imported by
XMLImport
Contents: 2013/07/01  19:22:39.07
 stamp\a{9318803d-1b46-486c-a59b-f7dcc92c4d2f}.xml imported by
XMLImport
Contents: 2013/07/01  19:22:39.07
 stamp\a{a258c50d-a15e-493b-a29d-cc0e782b5f54}.xml imported by
XMLImport
Contents: 2013/07/01  19:22:39.07

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

Status: +0001
Informative:  1[0] - indicated directory contains no documents
Press a key to terminate.
```

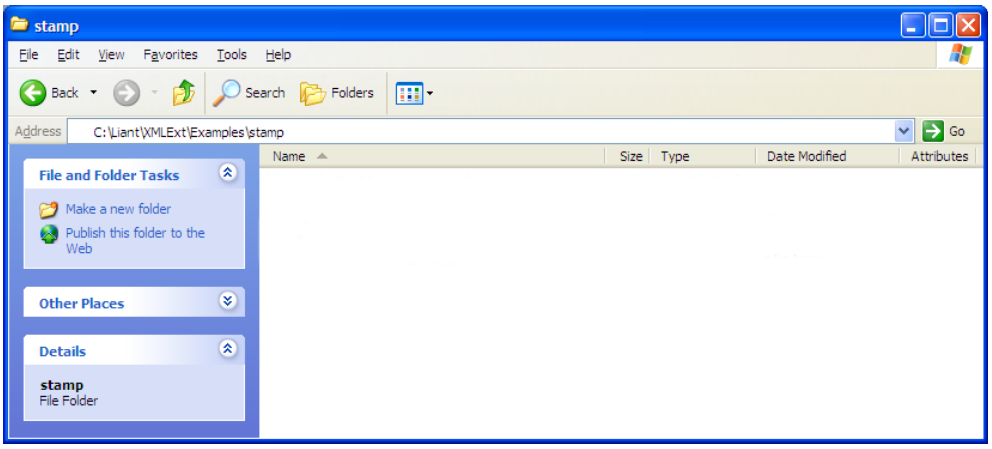
XML Document (Example 6)

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.


stamp
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Liant\XMLExt\Examples\stamp Go
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Other Places
Details
stamp
File Folder

Name	Size	Type	Date Modified	Attributes
a{3a2b7d60-6065-4785-bdf3-ab388992079d}.xml	1 KB	XML Document	8/19/2008 10:23 AM	A
a{4a4e8482-d3a4-492e-a79d-ec6d967cd4e6}.xml	1 KB	XML Document	8/19/2008 10:23 AM	A
a{30cd08ac-0edf-4885-a106-4acdcb8caada}.xml	1 KB	XML Document	8/19/2008 10:23 AM	A
a{9318803d-1b46-486c-a59b-f7dcc92c4d2f}.xml	1 KB	XML Document	8/19/2008 10:23 AM	A
a{a258c50d-a15e-493b-a29d-cc0e782b5f54}.xml	1 KB	XML Document	8/19/2008 10:23 AM	A

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


stamp
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Liant\XMLExt\Examples\stamp Go
Name Size Type Date Modified Attributes
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Other Places
Details
stamp
File Folder

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.

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

This example uses the following XML statements:

- XML INITIALIZE, which initializes or opens a session with XML Extensions.
- XML EXPORT FILE, which constructs an XML document (as a file) from the content of a COBOL data item.
- XML TEST WELLFORMED-FILE, which verifies that an XML document conforms to XML syntax rules.

- XML VALIDATE FILE, which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML TERMINATE, which terminates or closes the session with XML Extensions.

Development for Example 7

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 7

Use the same commands as for example 1 except substitute example07 for example01.

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 Address-Struct (as defined in the copy file, s-struct.cpy) to an XML document with the filename of address07.xml using the XML EXPORT FILE statement.

Next, the syntax of address07.xml is verified using the XML TEST WELLFORMED-FILE statement.

Following this, the content of address07.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, Address-Struct is the same as in example 1.

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. All of this is the same as in previous examples.

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, example07.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
`Move … To Time-Stamp.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`   Address-Struct`
`   "address07"`
`   "Address-Struct".`
	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`
`    "address07".`
	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`
`    "address07"`
`    "example07".`
	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)

Same as example 1.

Termination Test Logic (Example 7)

Same as example 1.

Status Display Logic (Example 7)

Same as example 1.

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)

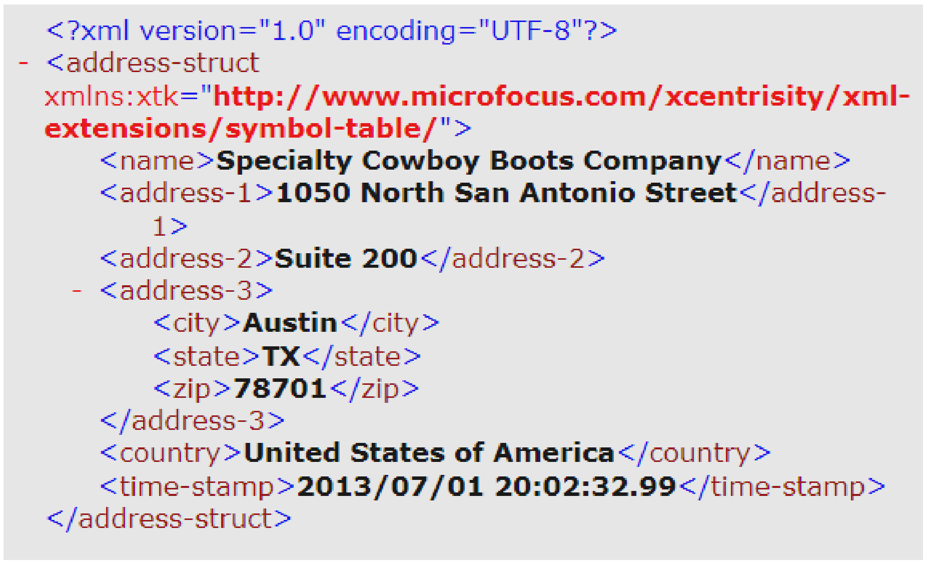
Note: Pressing a key will terminate the program.

Running the program (run example07) produces the following display:

```
Example-07 - Illustrate TEST-WELLFORMED FILE & VALIDATE FILE

address07.xml exported by XML EXPORT FILE

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                          TX78701
Country:      United States of America
Time-Stamp:   2013/07/01  20:02:32.99

address07.xml checked by XML TEST WELLFORMED-FILE

address07.xml validated by XML VALIDATE FILE

address07.xml imported by XML IMPORT FILE

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                          TX78701
Country:      United States of America
Time-Stamp:   2013/07/01  20:02:32.99

You may inspect 'address07.xml'

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

XML Document (Example 7)

Microsoft Internet Explorer may be used to view the generated XML document address07.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"?>
- <address-struct
  xmlns:xtk="http://www.microfocus.com/xcentrisity/xml-
  extensions/symbol-table/">
      <name>Specialty Cowboy Boots Company</name>
      <address-1>1050 North San Antonio Street</address-
          1>
      <address-2>Suite 200</address-2>
    - <address-3>
          <city>Austin</city>
          <state>TX</state>
          <zip>78701</zip>
      </address-3>
      <country>United States of America</country>
      <time-stamp>2013/07/01 20:02:32.99</time-stamp>
  </address-struct>
```

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.

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

This example uses the following XML statements:

- XML INITIALIZE, which initializes or opens a session with XML Extensions.
- XML EXPORT TEXT, which constructs an XML document (as a text string) from the content of a COBOL data item.
- XML TEST WELLFORMED-TEXT, which verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE TEXT, which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML PUT TEXT, which copies an XML document from a text string to a data file.
- XML FREE TEXT, which releases the memory that was allocated by XML EXPORT TEXT to hold the XML document as a text string.
- XML TERMINATE, which terminates or closes the session with XML Extensions.

Development for Example 8

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 8

Use the same commands as for example 1 except substitute example08 for example01.

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 Address-Struct (as defined in the copy file, s-struct.cpy) to an in-memory XML document as defined by the variables, *Document-Pointer* and *Document-Length*, 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, Address-Struct, is the same as in example 1.

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. All of this is the same as in previous examples.

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, example08.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
`Move … To Time-Stamp.`	Populate the Time-Stamp field.

`XML EXPORT TEXT`
`    Address-Struct`
`    Document-Pointer`
`    Document-Length`
`    "Address-Struct".`
	Execute the XML EXPORT TEXT statement specifying:
	the data item address,
	the XML document text pointer,
	the XML document text length,
	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`
`    Document-Length.`
	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`
`   Document-Length`
`   "example08".`
	Execute the XML VALIDATE TEXT statement specifying:
	the XML document text pointer
	the XML document text length
	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`
`    Document-Length`
`    "address08".`
	Execute the XML PUT TEXT statement specifying:
	the XML document text pointer
	the XML document text length
	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)

Same as example 1.

Termination Test Logic (Example 8)

Same as example 1.

Status Display Logic (Example 8)

Same as example 1.

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)

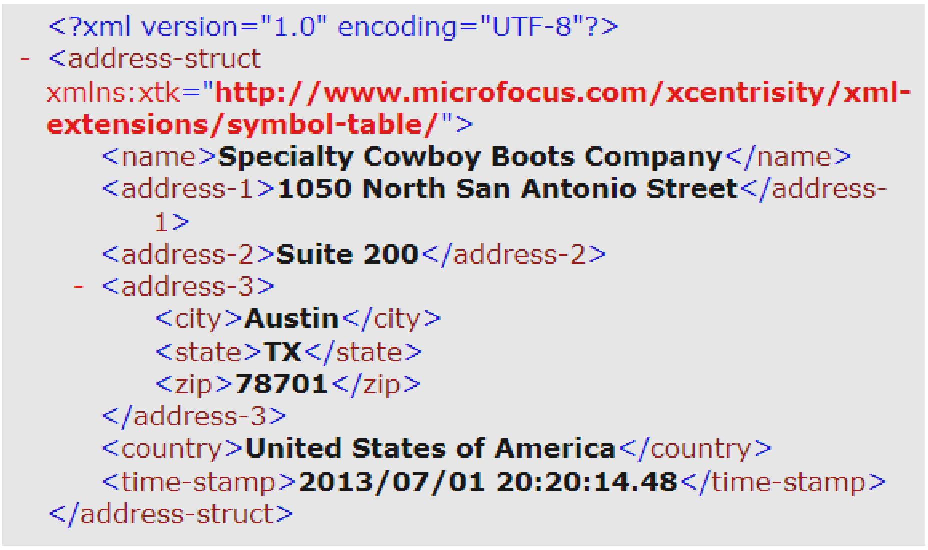
Note: Pressing a key will terminate the program.

Running the program (run example08) produces the following display:

```
Example-08 - Illustrate TEST-WELLFORMED TEXT & VALIDATE TEXT

Document exported by XML EXPORT TEXT

Name:        Specialty Cowboy Boots Company
Address-1:   1050 North San Antonio Street
Address-2:   Suite 200
Address-3:   Austin                         TX78701
Country:     United States of America
Time-Stamp:  2013/07/01  20:20:14.48

Document checked by XML TEST WELLFORMED-TEXT

Document validated by XML VALIDATE TEXT

Document memory imported by XML IMPORT TEXT

Name:        Specialty Cowboy Boots Company
Address-1:   1050 North San Antonio Street
Address-2:   Suite 200
Address-3:   Austin                         TX78701
Country:     United States of America
Time-Stamp:  2013/07/01  20:20:14.48

Document memory written by XML PUT TEXT

Document memory released by XML FREE TEXT

address08.xml imported by XML IMPORT FILE

Name:        Specialty Cowboy Boots Company
Address-1:   1050 North San Antonio Street
Address-2:   Suite 200
Address-3:   Austin                         TX78701
Country:     United States of America
Time-Stamp:  2013/07/01  20:20:14.48

You may inspect 'address08.xml'

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

XML Document (Example 8)

Microsoft Internet Explorer may be used to view the generated XML document, address08.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"?>
- <address-struct
  xmlns:xtk="http://www.microfocus.com/xcentrisity/xml-
  extensions/symbol-table/">
      <name>Specialty Cowboy Boots Company</name>
      <address-1>1050 North San Antonio Street</address-
          1>
      <address-2>Suite 200</address-2>
    - <address-3>
          <city>Austin</city>
          <state>TX</state>
          <zip>78701</zip>
      </address-3>
      <country>United States of America</country>
      <time-stamp>2013/07/01 20:20:14.48</time-stamp>
  </address-struct>
```

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 77) 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, which initializes or opens a session with XML Extensions.
- XML EXPORT FILE, which constructs an XML document (as a file) from the contents of a COBOL data item.
- XML IMPORT FILE, which reads an XML document (from a file) into a COBOL data item.
- XML TRANSFORM FILE, which uses an XSLT stylesheet to modify (transform) an XML document into another format.
- XML TERMINATE, which terminates or closes the session with XML Extensions.

Development for Example 9

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 9

Use the same commands as for example 1 except substitute example09 for example01.

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 Address-Struct (as defined in the copy file, s-struct.cpy) to an XML document with the filename of address09a.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, address09b.xml, and then transformed back into the original output format.

Next, the content of the XML document is imported from the file, address09c.xml, and placed in the same data item using the XML IMPORT FILE statement.

Subsequently, the content of the XML document, address09c.xml, is transformed using the **example09.xsl** stylesheet to create the file, address09d.xml. This adds all of the COBOL attributes to address09d.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, Address-Struct, is the same as in example 1.

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. All of this is the same as in previous examples.

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, example09.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
`Move … To Time-Stamp.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`    Address-Struct`
	Execute the XML EXPORT FILE statement specifying:
	the data item address,

`"address09a"` `"Address-Struct".`	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`
`"address09a"`
`"toext"`
`"address09b".`
	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
`Initialize Address-Struct.`	Ensure that the address item contains no data.

`XML TRANSFORM FILE`
`"address09b"`
`"toint"`
`"address09c".`
	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`
`Address-Struct`
`"address09c"`
`"Address-Struct".`
	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`
`"address09c"`
`"example09"`
`"address09d".`
	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)

Same as example 1.

Termination Test Logic (Example 9)

Same as example 1.

Status Display Logic (Example 9)

Same as example 1.

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.

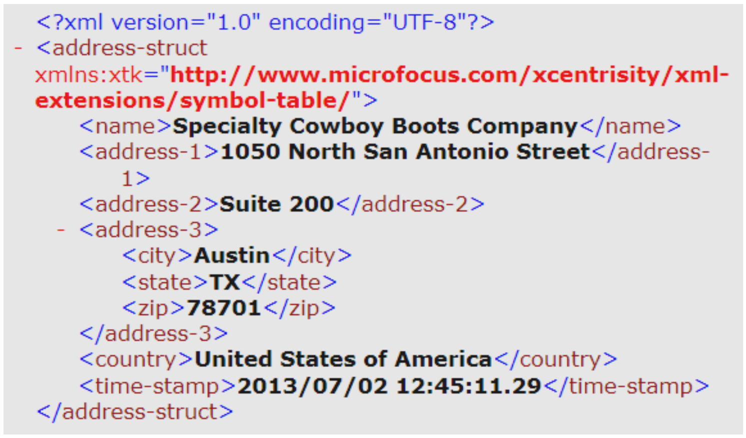
Running the program (run example09) produces the following display:

```
Example-09 - Illustrate TRANSFORM FILE

address09a.xml exported by XML EXPORT FILE


Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                         TX78701
Country:      United States of America
Time-Stamp:   2013/07/02  12:45:11.29

address09a.xml transformed into address09b.xml by XML TRANSFORM
FILE

address09b.xml transformed into address09c.xml by XML TRANSFORM
FILE

address09c.xml imported by XML IMPORT FILE

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                         TX78701
Country:      United States of America
Time-Stamp:   2013/07/02  12:45:11.29

address09c.xml transformed into address09d.xml by XML TRANSFORM
FILE

You may inspect 'address09a.xml' - 'address09d.xml'

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

XML Documents (Example 9)

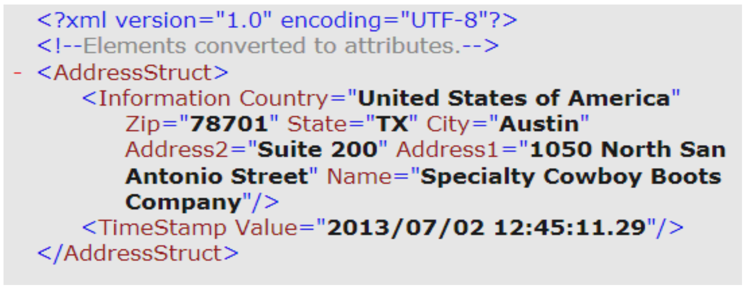
Microsoft Internet Explorer may be used to view the generated XML documents, address09a.xml, address09b.xml, address09c.xml, and address09d.xml. Their content of these documents should appear as follows. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

address09a.xml – Internal Format (similar to address01.xml)



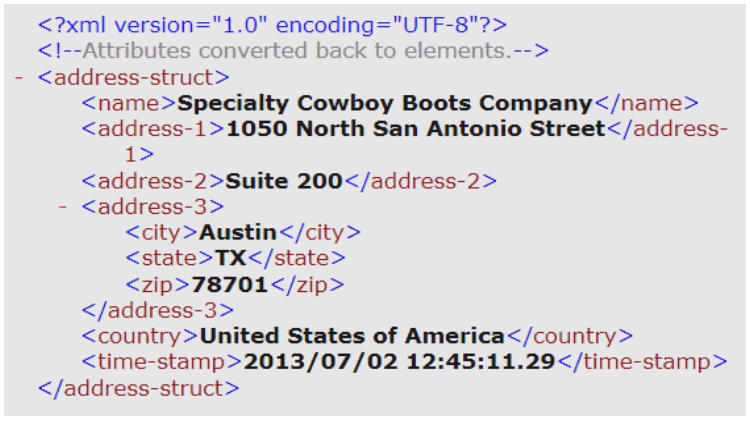
```
   <?xml version="1.0" encoding="UTF-8"?>
 - <address-struct
   xmlns:xtk="http://www.microfocus.com/xcentrisity/xml-
   extensions/symbol-table/">
       <name>Specialty Cowboy Boots Company</name>
       <address-1>1050 North San Antonio Street</address-
           1>
       <address-2>Suite 200</address-2>
     - <address-3>
           <city>Austin</city>
           <state>TX</state>
           <zip>78701</zip>
       </address-3>
       <country>United States of America</country>
       <time-stamp>2013/07/02 12:45:11.29</time-stamp>
   </address-struct>
```

address09b.xml – External Format (similar to address02.xml)



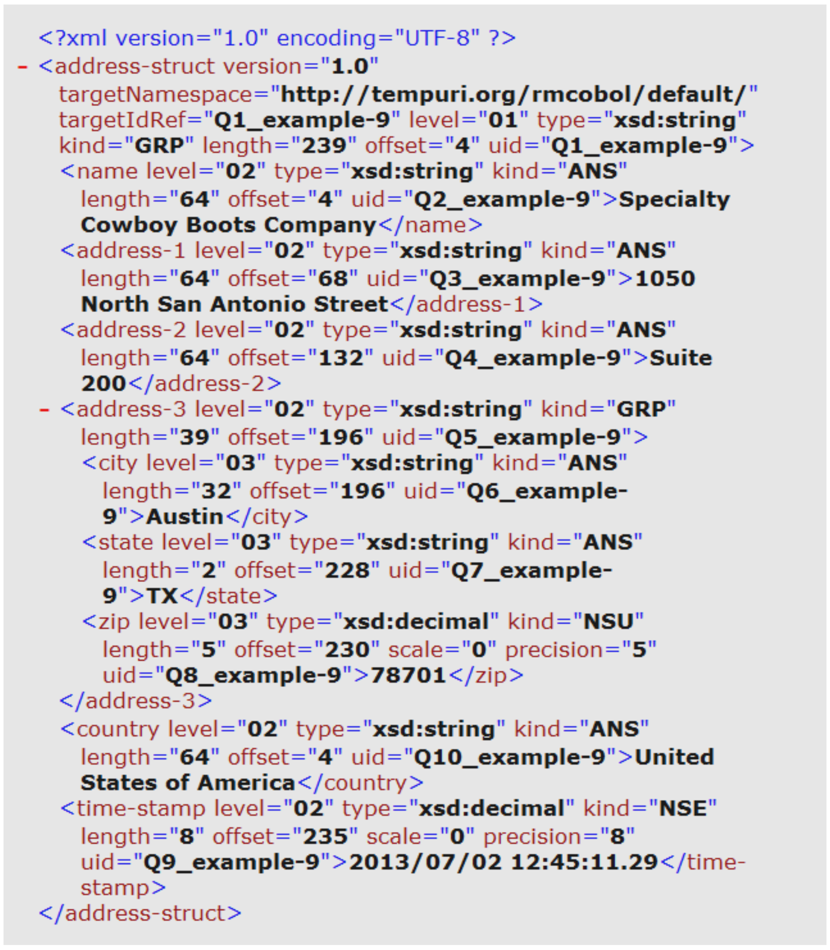
```
   <?xml version="1.0" encoding="UTF-8"?>
   <!--Elements converted to attributes.-->
 - <AddressStruct>
       <Information Country="United States of America"
           Zip="78701" State="TX" City="Austin"
           Address2="Suite 200" Address1="1050 North San
           Antonio Street" Name="Specialty Cowboy Boots
           Company"/>
       <TimeStamp Value="2013/07/02 12:45:11.29"/>
   </AddressStruct>
```

address09c.xml – Internal Format Restored



```
   <?xml version="1.0" encoding="UTF-8"?>
   <!--Attributes converted back to elements.-->
 - <address-struct>
       <name>Specialty Cowboy Boots Company</name>
       <address-1>1050 North San Antonio Street</address-
           1>
       <address-2>Suite 200</address-2>
     - <address-3>
           <city>Austin</city>
           <state>TX</state>
           <zip>78701</zip>
       </address-3>
       <country>United States of America</country>
       <time-stamp>2013/07/02 12:45:11.29</time-stamp>
   </address-struct>
```

address09d.xml – Internal Format plus COBOL Attributes



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <address-struct version="1.0"
    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">Specialty
      Cowboy Boots Company</name>
    <address-1 level="02" type="xsd:string" kind="ANS"
      length="64" offset="68" uid="Q3_example-9">1050
      North San Antonio Street</address-1>
    <address-2 level="02" type="xsd:string" kind="ANS"
      length="64" offset="132" uid="Q4_example-9">Suite
      200</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">78701</zip>
    </address-3>
    <country level="02" type="xsd:string" kind="ANS"
      length="64" offset="4" uid="Q10_example-9">United
      States of America</country>
    <time-stamp level="02" type="xsd:decimal" kind="NSE"
      length="8" offset="235" scale="0" precision="8"
      uid="Q9_example-9">2013/07/02 12:45:11.29</time-
      stamp>
  </address-struct>
```

Example 10: 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, which initializes or opens a session with XML Extensions.
- XML TEST WELLFORMED-FILE, which verifies that an XML document conforms to XML syntax rules.
- XML VALIDATE FILE, which verifies that the content of an XML document conforms to rules specified by an XML schema file.
- XML TERMINATE, which terminates or closes the session with XML Extensions.

Development for Example 10

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 10

Use the same commands as for example 1 except substitute example10 for example01.

Program Description for Example 10

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 x-addressa1.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 x-addressa2.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 x-addressa3.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 10

The content of the COBOL data item, Address-Struct, is the same as in example 1.

Other Definitions for Example 10

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. All of this is the same as in previous examples.

Program Structure for Example 10

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, example10.cbl.

Initialization (Example 10)

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 10)

COBOL Statement	**Description**

`XML TEST WELLFORMED-FILE "x-addressa1".`	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 "x-addressa2".`	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 10)

COBOL Statement	**Description**
`XML VALIDATE FILE "x-addressa2" "example10".`	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 10)

COBOL Statement	**Description**
`XML TEST WELLFORMED-FILE "x-addressa3".`	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 10)

COBOL Statement	**Description**
`XML VALIDATE FILE "x-addressa3" "example10".`	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 10)

Same as example 1.

Termination Test Logic (Example 10)

Same as example 1.

Status Display Logic (Example 10)

Same as example 1.

Execution Results for Example 10

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

COBOL Display (Example 10)

Running the program (run example10) 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 10 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-10 - Illustrate diagnostics for invalid documents
 and documents that are not well-formed

XML TEST WELLFORMED-FILE - not well-formed ...
Error: 28[34] - in function: LoadDocument
MSXML 6.0 parse error code: 0xc00cee3b
The name in the end tag of the element must match the element type
in the start\
 tag.
line 13, position 21
</non-address-struct> <!-- intentional tag mismatch -->
--------------------|
C:\xmlext\examples\x-addressa1.xml
HRESULT: 0x80004005
  ... wellformedness test failed (as it should).

Press a key to continue:
```

For UNIX, the first display would be shown as follows:

```
Example-10 - 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
/usr/xmlext/examples/x-addressa1.xml
13: parser error : Opening and ending tag mismatch: address-struct
line 2 and n\
on-address-struct
</non-address-struct> <!-- intentional tag mismatch -->
                     ^
  ... wellformedness test failed (as it should).

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 ...
  ... wellformedness test succeeeded.

XML VALIDATE FILE - well-formed - invalid ...
Error: 28[34] - in function: LoadDocument
MSXML 6.0 parse error code: 0xc00ce201
```

```
Error parsing 'ABCDE' as decimal datatype. The element 'zip' with
value 'ABCDE\
' failed to parse.
line 9, position 20
    <zip>ABCDE</zip>  <!-- intentional nonnumeric zip code -->
-------------------|
C:\xmlext\examples\x-addressa2.xml
HRESULT: 0x80004005
  ... validation failed (as it should).

Press a key to continue:
```

For UNIX, the second display would be shown as follows:

```
XML TEST WELLFORMED-FILE - well-formed & invalid ...
  ... wellformedness test succeeeded.

XML VALIDATE FILE - well-formed - invalid ...
Error: 28[4] - in function: LoadDocument
/usr/xmlext/examples/x-addressa2.xml
9: element zip: Schemas validity error : Element 'zip': 'ABCDE' is
not a valid \
value of the atomic type 'type_Q8_example-a'.
  ... validation failed (as it should).

Press a key to continue:
```

Third Display

Note: Pressing a key will terminate the program.

```
XML TEST WELLFORMED-FILE - well-formed - valid ...
  ... wellformedness test succeeeded.

XML VALIDATE FILE - well-formed - valid ...
  ... validation successful.

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 11: 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 11 illustrates this situation more fully.

This example uses the following XML statements:

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

Development for Example 11

The COBOL program must be compiled with an Enterprise Developer compiler using the XMLGEN compiler directive to generate an XML-format model file.

After the successful compilation, you may then execute the COBOL program.

Batch File for Example 11

Use the same commands as for example 1 except substitute example11 for example01.

Program Description for Example 11

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 Address-Struct (as defined in the copy file, s-struct.cpy) to an XML document with the filename of address11.xml using the XML EXPORT FILE statement.

Next, the content of the XML document is imported from the file, address11.xml, and placed in the same data item using the XML IMPORT FILE statement.

Additionally, the content of the predefined XML document named x-address11.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 11

The content of the COBOL data item, Address-Struct, is the same as in example 1.

Other Definitions for Example 11

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. All of this is the same as in previous examples.

Program Structure for Example 11

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, example11.cbl.

Initialization (Example 11)

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 11)

COBOL Statement	**Description**
`Move … To Time-Stamp.`	Populate the Time-Stamp field.

`XML EXPORT FILE`
`    Address-Struct`
`    "address11"`
`    "Address-Struct".`
	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 11)

COBOL Statement	Description
`Initialize Address-Struct.`	Ensure that the address item is initialized.
`XML IMPORT FILE Address-Struct "address11" "Address-Struct".`	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 11)

COBOL Statement	Description
`Initialize Address-Struct.`	Ensure that the address item is initialized.
`XML IMPORT FILE Address-Struct "x-address11" "Address-Struct".`	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 11)

Same as example 1.

Termination Test Logic (Example 11)

Same as example 1.

Status Display Logic (Example 11)

Same as example 1.

Execution Results for Example 11

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

COBOL Display (Example 11)

Note: Pressing a key will terminate the program.

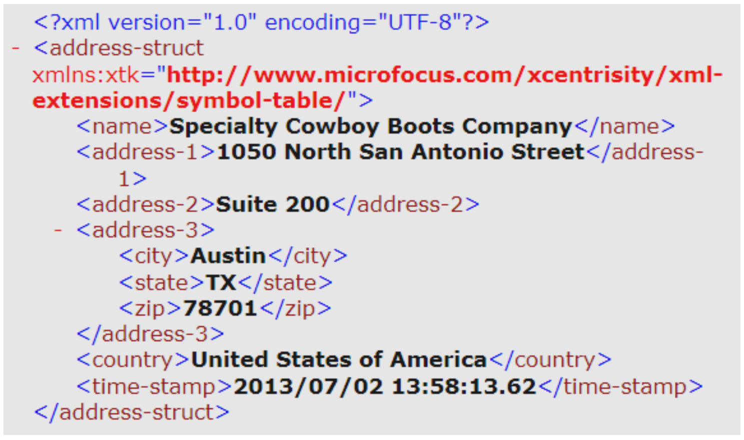
Running the program (run example11) produces the following display:

```
Example-11 - Illustrate IMPORT with missing intermediate names

address11.xml exported by XML EXPORT FILE

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                          TX78701
Country:      United States of America
Time-Stamp:   2013/07/02  13:58:13.62

address11.xml imported by XML IMPORT FILE:

Name:         Specialty Cowboy Boots Company
Address-1:    1050 North San Antonio Street
Address-2:    Suite 200
Address-3:    Austin                          TX78701
Country:      United States of America
Time-Stamp:   2013/07/02  13:58:13.62

x-address11.xml imported by XML IMPORT FILE:

Name:         Wild Hair Corporation
Address-1:    3543 Hair Court
Address-2:    Sweet 800
Address-3:    Lostin                          TX78787
Country:      United States of America
Time-Stamp:   00:00:00.00

You may inspect 'address11.xml' & 'x-address11.xml'

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

XML Document (Example 11)

Microsoft Internet Explorer may be used to view the generated XML document, address11.xml and the predefined XML document x-address11.xml. (Note that Internet Explorer will differentiate among the various syntactical elements of XML by displaying them in different colors.)

address11.xml



```
  <?xml version="1.0" encoding="UTF-8"?>
- <address-struct
  xmlns:xtk="http://www.microfocus.com/xcentrisity/xml-
  extensions/symbol-table/">
     <name>Specialty Cowboy Boots Company</name>
     <address-1>1050 North San Antonio Street</address-
        1>
     <address-2>Suite 200</address-2>
   - <address-3>
        <city>Austin</city>
        <state>TX</state>
        <zip>78701</zip>
     </address-3>
     <country>United States of America</country>
     <time-stamp>2013/07/02 13:58:13.62</time-stamp>
  </address-struct>
```

x-address11.xml

```
  <?xml version="1.0" encoding="UTF-8"?>
- <address-struct>
      <name>Wild Hair Corporation</name>
      <address-1>3543 Hair Court</address-1>
      <address-2>Sweet 800</address-2>
      <city>Lostin</city>
      <state>TX</state>
      <zip>78787</zip>
      <country>United States of America</country>
      <time-stamp>00:00:00.00</time-stamp>
  </address-struct>
```

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 example*.int del example*.int
if exist example*.exe del example*.exe
if exist example*.lst del example*.lst
if exist example*.xml del example*.xml
if exist address*.xml del address*.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 stamp\*.xml rmdir stamp /q /s
echo                            finished cleanup.
```

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

```
cleanup
```

example.bat

This batch file will compile a COBOL source program, and execute the COBOL program. The content of this file is as follows:

```
@echo off
setlocal
REM %1 == example program file name (without extension)

REM Compile the example program given by %1
set COBCPY=..;%COBCPY%
cobol %1 list() xmlgen(ws) noobj;
REM To compile for .NET managed code comment out the above
line and uncomment the following line
REM cobol %1 list() xmlgen(ws) ilgen iltarget(x86)
```

```
ilref(MicroFocus.COBOL.XmlExtensions)
ilref(MicroFocus.COBOL.XmlExtensions.Interop);

REM Run example program
REM    COBPATH needs to include the location of cobxmlext.dll.
run %1
endlocal
```

This batch file uses parameters that are specified by the caller of the batch file. The parameter is the filename of the COBOL program (without the .cbl extension).

To build and run *Example 1: Export File and Import File* (on page 72) using this batch file, enter the following on the command line:

```
example example01
```

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 Example01 - Export / Import File.
call example example01
echo Example02 - Export / Import with XSLT stylesheets.
call example example02
echo Example03 - Export / Import with Occurs Depending.
call example example03
echo Example04 - Export / Import with sparse arrays.
call example example04
echo Example05 - Export / Import Text.
call example example05
echo Example06 - Export / Import with directory polling.
mkdir stamp
call example example06
echo Example07 - Export / Well-Formed File / Validate File.
call example example07
echo Example08 - Export / Well-Formed Text / Validate Text.
call example example08
echo Example09 - Export / Transform / Import.
call example example09
echo Example10 - Well-Formed / Validate diagnostics.
call example example10
echo Example11 - Import with missing intermediate names.
call example example11
```

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.sh is provided for the same purpose as examples.bat on Windows.

8. 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. Click on the Samples link installed by Enterprise Developer to use the Samples Browser to locate the XML Extensions samples and examples.

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.

9. 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 130 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 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**
-5	Warning – ambiguous model data name	More than one matching data name was found in the document. The most recently found data name is used.
-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.

Message Number	**Severity and Message Text**	**Description**
3	Warning – internal logic – memory not deallocated	During process cleanup, memory blocks that should have already been deallocated were still allocated.
9	Error – in function: CreateDocument	The underlying XML parser detected an error while trying to create an XML document.
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.
12	Error – data item – not found	The xmlif library 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.
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.
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.
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.
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.
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 the model files produced by the XMLGEN compiler directive were not re-generated and are outdated.

Message Number	**Severity and Message Text**	**Description**
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). Occasionally, XML Extensions generates documents that are then loaded as input documents. In the unlikely event that the generated document contains errors, a load document error will be encountered.
29	Error – in function: LoadSchema	An error was detected while trying to load an XML schema file. This normally means that there was a problem locating the document (either the document does not exist or there is a problem with permissions) or that the schema itself is in error.
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.
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.
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).
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.
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.
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 model file	The symbol table was not found in the specified model file. The model file was probably not produced by the XMLGEN compiler directive.

Message Number	**Severity and Message Text**	**Description**
42	Error - in function: WriteDocument	An error occurred while attempting to write an XML document from the internal Document Object Model representation.
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 MF_XMLEXT_LOCAL_ ENCODING value	The value of the MF_XMLEXT_LOCAL_ENCODING environment variable on UNIX is not any of "mflatin1", "mflatin9", or a name recognized by the available iconv library.
48	Error - unable to locate iconv library	The value of the MF_XMLEXT_LOCAL_ENCODING environment variable on UNIX is not "mflatin1" nor "mflatin9", 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 Enterprise Developer is installed on Windows, but could not be found. MSXML 6.0 can be obtained by downloading it from Microsoft's web site.
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 - 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.
62	Error - requested template file cannot be found	Requested template file specified by the model file data name parameter cannot be found.

Message Number	**Severity and Message Text**	**Description**
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. 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.
68	Error – add attribute node failed	Adding an attribute to the document failed.
69	Error – add element node failed	Adding an element to the document failed.
70	Error – add text node failed	Adding a text mode to the document failed.
71	Error – validate DOM document	Validating the DOM document failed.
72	Error – incorrect XML statement parameters	One or more of the parameters passed into the XML statement are invalid.

10. 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. The Enterprise Developer compiler's XMLGEN directive generates a file containing an XML representation of a 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 FILE/TEXT statements. See also *XSLT stylesheet* (on page 137).

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 136), *XHTML* (on page 137), and *XML* (on page 137).

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 MF_XMLEXT_LOCAL_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”.

Model files

XML document files created by the XMLGEN compiler directive. The model file is usually named the same as the COBOL program source file but with an extension of **.xml**.

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 135) and *XML* (on page 137).

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 137).

Stylesheet

See *XSLT stylesheet* (on page 137).

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 area 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 136).

Valid XML document

See *Schema valid XML document* (on page 136).

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 135).

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 136).

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 137) and *W3C* (on page 138).

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 137) and *W3C* (on page 138).

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.

11. Index