



XML Guide for PERCobol

The contents of this manual may be revised without prior notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical for any purpose without the expressed written permission of LegacyJ Corporation.

Second Edition: April, 2003

LegacyJ has made every effort to ensure that this manual is correct and accurate, but reserves the right to make changes without notice at its sole discretion at any time.

Preface

XML, the Extensible Markup Language, is a standardized document format. Unlike the majority of document formats, it is not proprietary, it is equally suited to hierarchical or narrative documents, and it is widely supported by a range of software and industries. PERCobol now supports XML directly using the same file statements available for sequential input files.

The goal of PERCobol XML is to allow modifications to be made to the file-control and file section to allow an XML document to directly replace an input sequential file. Only additional file status codes are added and meaningless file status codes removed.

ORGANIZATION XML files currently are for input only. XML currently does not define a special output mode; use a line sequential output file to create XML.

Contents

XML BACKGROUND	1
STRUCTURE OF XML	2
PARSING XML	4
XML IN PERCOBOL	5
CONTENT	8
ATTRIBUTES	9
LOCATOR	10
ASSIGN TO FOR XML	11
XML FILE ERROR INFORMATION	12
XML DEPLOYMENT	13

XML Background

XML is a descendent of SGML, like HTML, but more rigorously implemented than HTML without the complexities of SGML. XML is human-readable, with all data stored in text format. XML is machine-readable, with enforceable structure.

XML has nothing but custom tags, tags invented for a particular purpose. For a card catalog, the tags may include title, publisher, and description. For a customer list, the data may include the customer name, the date of first and last purchase, and the average purchase price. Each of these data items is tagged. New information may be added later without breaking the existing format. Some basic validation of structure may be built into the file itself, enforced automatically upon read.

An XML application is not a program, but a set of tags and attributes, rigorously defined. One XML application is XHTML, a more rigorous version of HTML. Other XML applications exist for a large number of different industries and needs. An XML application may already exist for the industry or need in question, or a new application may be developed by defining the tags and attributes necessary.

Structure of XML

An XML file looks like the following narrative document:

```
<?xml version="1.0" encoding="US-ASCII" standalone="yes">
<biography>
<person scientist="yes">Albert Einstein</person> discovered relativity.
</biography>
```

or the following hierarchical document.

```
<?xml version="1.0" encoding="US-ASCII" standalone="yes">
<customer-database>
  <customer>
    <first-name>Jack</first-name>
    <last-name>Smith</last-name>
    <address-1>123 Elm St.</address-1>
    <city>Springfield</city>
    <state>IL</state>
    <zip>12345</zip>
  </customer>
  <customer>
    <first-name>Jill</first-name>
    <last-name>White</last-name>
    <address-1>234 Maple St.</address-1>
    <city>Springfield</city>
    <state>IL</state>
    <zip>12345</zip>
  </customer>
</customer-database>
```

The first line of the file indicates with a processing instruction, starting with an angle bracket then question mark, that this is an XML file, and the encoding for the file, and whether this file stands alone. This XML processing line is not absolutely required for all files, but it should always be present.

In both cases, the tags indicated by <> characters markup the document with a description of the content. The content is between the tags, such as Albert Einstein or Springfield. An attribute is a name/value pair within a tag, such as scientist = "yes".

Unlike HTML, each tag must have an end tag, beginning with `</`, or be marked as both a start and end tag by including the `/` at the end of tag, like `<hr/>`. Each attribute must have a value, and the value must be in either single- or double-quotes; it cannot be left unquoted. Each document must have a root element, a single tag which surrounds the document data. (In XHTML, for instance, this is `<html>` surrounding all the HTML.)

Also unlike HTML, each tag and attribute is case-sensitive. A `<person>` is different from a `<Person>`.

The white space between tags in an XML file is generally ignored, but some parsers can recognize it.

Parsing XML

Parsing XML is the reading of XML, separating the structure from the contents, determining that the document is well-formed and possibly validating the contents against a DTD, a document type definition. A well-formed document follows all the rules of XML. A valid document follows all the rules of an XML application. (A document not well-formed cannot be valid.) A DTD can help ensure that an invalid document is recognized as such, but it cannot ensure that the data itself is valid. That is, it can ensure that a person has a birth-date child element, but it cannot ensure that the birth-date is not in the future when it's expected to be in the past; that is the job of the program handling the data.

The current generation of parsers can handle namespaces. Namespaces are the separation of XML application domains into separate spaces from one another. For example, this allows a single document to have a customer:description and a product:description without conflict. The customer: and product: are namespaces.

XML in PERCobol

XML in PERCobol is defined as a file with ORGANIZATION XML. An XML file is largely like a record sequential file in usage; the only access mode currently supported is SEQUENTIAL.

The tag, such as <person> is the basic structural element driving the remainder of the parsing. The tag has zero or more attributes, zero or more characters of data until the end-tag is reached, and the parser may recognize that it is at a certain location in the file.

The tag structure of the application and the attributes should be known before reading or writing the file. The values of the attributes and contents of the tags are the unknown that the READ can read.

Not all tags used in a document must be referenced from the program code. This allows a future revision of the document to include new tags without breaking the older programs; the new tags will simply be ignored. This allows a customer database to add a field giving the customer birth-date without interfering with the billing program; both a birthday card program and billing can run off the same file.

Any tags used must in the correct hierarchical order. That is, if a person has an attribute of birth-date, the attribute must be a child of person. If a person has character content describing the person, that the description must be a child of person. The hierarchical order is important; filling in pieces irrelevant to the program is not important.

An XML file does not directly have the concept of a record. PERCobol forms XML structures into records on the fly, initializing record items which are not defined within a particular record. PERCobol will form a new record when a second level (level below the document root) is defined. PERCobol will also form a new record when a tag is repeated within a record. Generally, data which a Cobol program would consider handling will be well-formed for record handling.

When reading an XML file, PERCobol tracks the tags in order. If <a><c val=1></c><c val=2></c> is found, then two records are found; one is <a><c val=1>, the second is <a><c val=2>. The higher levels of hierarchy are preserved; when a repetition is found, it is treated as a separate record. By treating the file in this fashion, normal Cobol records are generated; the Cobol code does not need to do separate tracking of the higher-level structural elements.

If records are found in the file that does not match the structure mentioned in the Cobol program, they are skipped. If attributes are found that do not match the structure mentioned in the Cobol program, they are skipped. By only reading in the data which is mentioned, the code is protected from changes in the file or file format.

An XML data record is highly dependent upon its structure, not its byte positions. Certain words are used to mark the structural elements of the XML record. The

hierarchical structure of the record is used to map to the hierarchical structure of the XML data.

The name of each data item is important and it must match the XML name exactly; use the IDENTIFIED BY clause to name it exactly. Remember that XML is case sensitive and dashes and underscores are considered separate characters.

Because the naming and record order is considered the important attribute of an XML record, do not redefine or implicitly redefine an XML record (multiple level 01). It is acceptable to make the content-data a group item containing data which is redefined.

The IDENTIFIED BY clause may include IDENTIFIED BY ANY to indicate that any text will match. Which tag was actually found will be returned in the content-name or attribute-name property.

Three forms are available for each of the data items available for an XML tag, a quick form, a short form and a long form. Any may be used, even within a single record; the long provides more information.

An example of the tag format is given below:

Quick Format

```
05 TAG [IDENTIFIED BY "tag-name"] PIC X(n) VALUE content-data.
```

Short format

```
05 TAG [IDENTIFIED BY "tag-name"] [PIC X(n) VALUE].  
10 PCDATA CONTENT PIC X(n) VALUE content-data.  
10 ATTRIBUTE-NAME ATTRIBUTE [IDENTIFIED BY "attribute-name"]  
    PIC X(n) VALUE attribute-data.  
10 LOCATOR-ITEM LOCATOR PIC X(n) VALUE locator-system.
```

Long format

```
05 TAG [IDENTIFIED BY {ANY | "tag-name"}] [PIC X(n) VALUE].  
10 PCDATA CONTENT.  
15 PCDATA-NAME CONTENT-NAME PIC X(n) VALUE content-name.  
    15 PCDATA-OFFSET-ITEM CONTENT-OFFSET PIC 9(n)  
        VALUE content-offset.  
    15 PCDATA-LENGTH-ITEM CONTENT-LENGTH PIC 9(n)  
        VALUE content-length.  
    15 PCDATA-DATA-ITEM CONTENT-DATA PIC X(n)  
        VALUE content-data.  
10 ATTR-NAME ATTRIBUTE [IDENTIFIED BY {ANY | "attribute-name"}]  
    15 ATTR-NAME ATTRIBUTE-NAME PIC X(n)  
        VALUE attribute-name.
```

```

15 ATTR-URI ATTRIBUTE-URI PIC X(n)
      VALUE attribute-uri.
15 ATTR-LOCAL ATTRIBUTE-LOCAL PIC X(n)
      VALUE attribute-local.
15 ATTR-QNAME ATTRIBUTE-QNAME PIC X(n)
      VALUE attribute-qname.
15 ATTR-TYPE ATTRIBUTE-TYPE PIC X(8) VALUE attribute-type.
15 ATTR-DATA ATTRIBUTE-DATA PIC X(n) VALUE attribute-data.
15 ATTR-LENGTH ATTRIBUTE-LENGTH PIC 9(n)
      VALUE attribute-length.
10 LOCATOR-GROUP LOCATOR.
    15 LOC-PUBLIC LOCATOR-PUBLIC VALUE locator-public.
    15 LOC-SYSTEM LOCATOR-SYSTEM VALUE locator-system.
    15 LOC-LINE LOCATOR-LINE VALUE locator-line.
    15 LOC-COL LOCATOR-COLUMN VALUE locator-column.

```

In each tag, everything in a record declared at a higher level than CONTENT is content information. Everything in a record declared at a higher level than ATTRIBUTE is attribute information. Everything in a record declared at a higher level than LOCATOR is locator information. Only one (1) LOCATOR per XML record is allowed.

CONTENT

The content in a tag is PCDATA, parsed character data. It is all the data between the start tag and the end tag, the actual content of the tag. It has any escape entities already replaced (such as '>' transformed into the greater than sign '>').

Because PCDATA is the most common information desired, all other information for a tag may be omitted; in such a case, the tag itself holds the data. This is sufficient for many simple XML parsing jobs. The following data record shows a simple XML record for reading names from a customer database.

```
01 CUSTOMER-DATABASE IDENTIFIED BY "customer-database".  
    05 CUSTOMER-NAME IDENTIFIED BY "name" PIC X(40).
```

This could read three records from the following XML file:

```
<?xml version="1.0">  
<customer-database>  
    <name>George Washington</name>  
    <name>John Adams</name>  
    <name>Thomas Jefferson</name>  
</customer-database>
```

When using IDENTIFIED BY ANY for the tag, use CONTENT-NAME to discover the name of the tag. This allows a more compact record description, more flexible in its input; this places more of the input burden on the user program.

Attributes

Attributes are items present within a tag. The attributes give information not within the narrative content. The attribute information is always a simple value within quotes. No structure is implied for the attribute itself; it merely describes the tag and content.

The attribute name must match that given in the text exactly. The attribute name is case sensitive. Because of this, the IDENTIFIED BY clause is used to give the exact name, including upper- or lower-case characters, underscores, hyphens, etc. IDENTIFIED BY ANY is acceptable for an attribute. IDENTIFIED BY "*"x" where x is a number starting at one (1) is also acceptable.

The ATTRIBUTE-VALUE is the value of the attribute. The ATTRIBUTE-VALUE-LENGTH is the actual length of the value; the PIC X allocated for ATTRIBUTE-VALUE must be long enough to hold the value. If an attribute is a list of tokens, then the tokens will be concatenated into a single piece of text with each token separated by a single space.

URI, the uniform resource identifier, is the namespace URI. The LOCAL-NAME is the local-name portion. The URI and LOCAL-NAME are only meaningful when namespaces are enabled.

QNAME is the fully qualified XML 1.0 name, the name as used before namespaces were conceived.

TYPE is the type of the attribute, always from one of the following: CDATA, ID, IDREF, IDREFS, NMTOKEN, NMTOKENS, ENTITY, ENTITIES, or NOTATION. CDATA is the default for an undeclared attribute.

When using IDENTIFIED BY ANY for the attribute, use ATTRIBUTE-NAME to discover the name of the attribute. This allows a more compact record description, more flexible in its input; this places more of the input burden on the program. A numbered tag may also be used, when IDENTIFIED BY "*#" where # is an integer value greater than or equal to zero. For example, IDENTIFIED BY "*3" will be the third attribute, whatever its name; its name will be discoverable through the ATTRIBUTE-NAME item.

Locator

The Locator is not guaranteed to be implemented by an XML parser. The Locator provides information as to where the tag or data was located. If not implemented by the XML parser, the data will be set to initialized data, SPACES or ZEROES. This information is generally not important for most uses. It should only be used for optional error reporting. Only one (1) LOCATOR per XML record is allowed.

LOCATOR-GROUP LOCATOR.

PUBLIC-ID LOCATOR-PUBLIC VALUE SPACES.

SYSTEM-ID LOCATOR-SYSTEM VALUE SPACES.

LINE-NUMBER LOCATOR-LINE VALUE ZEROES.

COLUMN-NUMBER LOCATOR-COLUMN VALUE ZEROES.

LOCATOR-ITEM LOCATOR PIC X(n) VALUE system-id.

ASSIGN TO for XML

The ASSIGN TO for XML is given by the following:

ASSIGN TO "xml:[xml-driver:][xml-feature...]xml-url"

The ASSIGN TO for an XML file specifies that the file is an XML file, what driver to use for the XML parser, and what features to enable or disable.

The driver name is the class name of an XML parser to use. A system default will be used if driver is not given.

The xml-url portion may be a local filename or a remote http: reference; it may be anything the driver accepts for a reader.

Feature is any of the following:

Xml-Feature	Property Name	Description
xml:	XML designator	optional, implied for XML organization
nons:	Namespaces:	namespaces are disabled
ns:	Namespaces:	namespaces are enabled (def)
nonsp:	Namespace-prefixes:	namespaces with prefix attributes are disabled (def)
nsp:	Namespace-prefixes:	namespaces with prefix attributes are enabled
noval:	Validation:	parser is not validating (no validate messages, def)
val:	Validation:	parser is validating (give validate messages)
nows:	Whitespace	ignore ignorable white space (def)
ws:	White space	report ignorable white space
noskip:	Skipped-Entity	ignore skipped-entity messages (def)
skip:	Skipped-Entity	give skipped-entity messages
nopc:	Processing-Code	ignore processing-code messages (def)
pc:	Processing-Code	give processing-code messages
noprefix:	start- and end-prefix	ignore prefix messages (def)
prefix:	start- and end-prefix	give prefix messages
elem:	element processing	give element processing messages
driver=name:	XML parser driver	specify SAX2 JAXP XML driver explicitly

XML File Error Information

XML allows up to three file status identifiers to be defined. The first is the standard two-digit file status code. The second and third are optional messages.

Not all file status codes are always returned. Some XML specific file codes are returned only when specifically enabled in the ASSIGN. The default set match what most Cobol programs expect.

General format

FILE STATUS numeric-error alphanumeric-file-status-2 alphanumeric-file-status-3

	FILE-STATUS-2	FILE-STATUS-3	Notes
00 OK (No Error)			; std
05 Open Optional Missing			; std, only on OPEN
07 Non-Reel requested, irrelevant for XML			; std, reel/unit
10 End Of File (EndDocument)			; std, only on READ
14 StartElement	uri	local-name / qname	; only if elem:
15 EndElement	uri	local-name / qname	; only if elem:
16 StartPrefixMapping	prefix	uri	; only if prefix
17 EndPrefixMapping	prefix		; only if prefix:
18 ProcessingInstruction	target	data	; only if pc
21 SkippedEntity	name		; only if skip:
11 Sax-warning	message		
20 Sax-error	message		
20 Parse error	message	locator-message	
30 Sax-fatal-error	message	locator-message	; std, error permanent
35 Open Non-Opt Missing	message	message	; std, only on OPEN
37 Open Mode N/S	message		; std, if cannot set features, load driver, etc.
39 Open Attr Mismatch	message		
41 Already open			; std, only on OPEN
42 Already closed			; std, only on CLOSE
46 Read already errored			; std, only on READ
47 Read not input	message		; std, only on READ

XML Deployment

XML parsing in PERCObol requires an external parser. This allows companies concentrating exclusively on XML to produce the best possible XML parsers that are then usable directly from PERCObol. One is supplied with PERCObol 2.7 and above for development, so a development environment requires nothing additional.

However, deployment may need an XML parser included if JAXP (Java API's for XML Processing) is not supported directly by the desired Java Virtual Machine (JVM). JAXP is an optional package for JDK 1.1.8 and higher. JAXP is included in Java 2 Standard Edition 1.4 and higher, and Java 2 Enterprise Edition 1.3 and higher. If not deploying to a JVM that already includes JAXP, be sure to deploy it with the application.

The current version of JAXP is available from Sun at http://java.sun.com/xml/xml_jaxp.html.

ORGANIZATION XML files also take an additional processing thread. Usually, this is not important for usage, but it can help parsing times dramatically on multi-processor systems. Also, some environments may forbid multiple threads; this would forbid using ORGANIZATION XML files. The XML thread is always a child thread of the COBOL thread doing the OPEN verb.