z/OS

XML System Services User's Guide and Reference

Version 2 Release 1

SA38-0681-00

Note

Before using this information and the product it supports, read the information in "Notices" on page 247.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 7. z/OS XML parser API: Assembler 111

Chapter 8. z/OS XML System Services exit interface. 145

Chapter 9. Diagnosis and problem determination 155

Appendix A. Return codes listed by value 159

Appendix B. Reason codes listed by value 161

Appendix C. xsdosrg command reference 213

Appendix D. C/C++ header files and assembler macros 215

Tables

About this document

This document presents the information you need to use the z/OS XML System Services (z/OS® XML) parser.

Who should use this document

This document is for application programmers, system programmers, and end users working on a z/OS system and using the z/OS XML parser.

This document assumes that readers are familiar with the z/OS system and with the information for z/OS and its accompanying products.

z/OS information

This information explains how z/OS references information in other documents and on the web.

When possible, this information uses cross document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see *z/OS Information Roadmap*.

To find the complete z/OS library, including the z/OS Information Center, see z/OS Internet Library (http://www.ibm.com/systems/z/os/zos/bkserv/).

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the "Contact us" web page for z/OS (http://www.ibm.com/systems/z/os/zos/webqs.html).
3. Mail the comments to the following address:

   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   US
4. Fax the comments to us, as follows:

   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:

  z/OS V2R1.0 XML User's Guide and Reference
  SA38-0681-00
- The topic and page number that is related to your comment.
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one of the following actions:

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at z/OS support page (http://www.ibm.com/systems/z/support/).

z/OS Version 2 Release 1 summary of changes

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Chapter 1. Introduction

What is XML?

XML allows you to tag data in a way that is similar to how you tag data when creating an HTML file. XML incorporates many of the successful features of HTML, but was also developed to address some of the limitations of HTML. XML tags may be user-defined, by either a DTD or a document written in the XML Schema language, that can be used for validation. In addition, namespaces can help ensure you have unique tags for your XML document. The syntax of XML has more restrictions than HTML, but this results in faster and cheaper browsing. The ability to create your own tagging structure gives you the power to categorize and structure data for both ease of retrieval and ease of display. XML is already being used for publishing, as well as for data storage and retrieval, data interchange between heterogeneous platforms, data transformations, and data displays. As these XML applications evolve and become more powerful, they may allow for single-source data retrieval and data display.

The benefits of using XML vary but, overall, marked-up data and the ability to read and interpret that data provide the following benefits:

- With XML, applications can more easily read information from a variety of platforms. The data is platform-independent, so now the sharing of data between you and your customers can be simplified.
- Companies that work in the business-to-business (B2B) environment are developing DTDs and schemas for their industry. The ability to parse standardized XML documents gives business products an opportunity to be exploited in the B2B environment.
- XML data can be read even if you do not have a detailed picture of how that data is structured. Your clients will no longer need to go through complex processes to update how to interpret data that you send to them because the DTD or schema gives the ability to understand the information.
- Changing the content and structure of data is easier with XML. The data is tagged so you can add and remove elements without impacting existing elements. You will be able to change the data without having to change the application.

However, despite all the benefits of using XML, there are some things to be aware of. First of all, working with marked up data can be additional work when writing applications because it physically requires more pieces to work together. Given the benefits of using XML, this additional work up front can reduce the amount of work needed to make a change in the future. Second, although it is a recommendation developed by the World Wide Web Consortium (W3C®), XML, along with its related technologies and standards including Schema, XPath, and DOM/SAX APIs, is still a developing technology.

An XML parser is a processor that reads an XML document and determines the structure and properties of the data. It breaks the data up into discrete units and provides them to other components. There are two basic types of XML parsers: non-validating and validating. A non-validating parser checks if a document is well-formed, but does not check a document against any DTDs or XML Schemas. A validating parser not only checks if a document is well-formed, but also verifies that it conforms to a specific DTD or XML Schema.

z/OS XML System Services

z/OS XML System Services (z/OS XML) is an XML processing component of the z/OS operating system. It contains an XML parser intended for use by system components, middleware, and applications that need a simple, efficient, XML parsing solution. z/OS XML can parse documents either with or without validation.

Note: The use of the term z/OS XML parser in this document refers specifically to the z/OS XML System Services parser.

The following are some distinct characteristics of z/OS XML:

- z/OS XML is an integrated component of z/OS. There is no need to download or install any additional packages to use it.
- z/OS XML provides a collection of programming interfaces for callers to use:
  - C/C++ and assembler interfaces to the z/OS XML parser itself.
  - C/C++, Java™, and UNIX command line interfaces to utility functions that build binary artifacts required for validation during a parse.

    Note: More information on the Java interfaces can be found in the Javadoc located at `/usr/include/java_classes/gxljdocs.jar`.
  - Assembler interfaces for user exits that give callers control over how the z/OS XML parser interacts with the rest of z/OS.
  - C/C++ interfaces to a service similar to a user exit, called a StringID handler, that allows for shorthand communications between the z/OS XML parser and the caller.
- The z/OS XML parser utilizes a buffer-in, buffer-out processing model instead of the event driven model common to SAX parsers. Input to, and output from the parser may span multiple buffers, allowing the caller to request parses for documents that are arbitrarily long.
- z/OS XML has minimal linkage overhead in order to reduce CPU usage as much as possible.
- z/OS XML provides assistive aids to the user in debugging not-well-formed documents.
- z/OS XML supports a number of character encodings, among them UTF-8, UTF-16 (big endian), IBM-1047 and IBM-037. There is no need on the part of the caller to transcode documents to a canonical encoding before calling the z/OS XML parser. For a full list of these supported encodings, see Appendix I, “Supported encodings,” on page 239.
- Support for enhanced error information records on a validating parse, see “Obtaining additional error information” on page 20.
- On a non-validating parse, support for toleration of an undefined prefix on an element or attribute. See “Support for undefined namespace prefix toleration” on page 12.

The z/OS XML parser is invoked as a callable service and can be used as such. The callable services stubs are shipped in CSSLIB.

Note about constant names: Some constant names begin with the string "GXLH". These constants are used solely by C callers. For assembler callers, remove the "GXLH" portion to get the appropriate constant name.

Chapter 2. Overview of z/OS XML System Services

This chapter provides an overview of the z/OS XML System Services; it briefly describes some of the XML features supported by the z/OS XML System Services and other technologies used by the z/OS XML parser. The following topics are discussed within this chapter:

- "z/OS XML System Services features"
- "z/OS XML System Services functions" on page 4
- "Document processing model" on page 6
- "Output buffer format" on page 8
- "Optimized Schema Representation" on page 8
- "String Identifiers" on page 8
- "Memory management" on page 8
- "Enable offload to specialty engines" on page 8

z/OS XML System Services features

The following is a list of features provided by z/OS XML System Services. References to additional information on the various features are provided where appropriate:

- An external C and C++ API, see "z/OS XML XL C/C++ API" on page 54
- An external assembler API, see Chapter 7, "z/OS XML parser API: Assembler," on page 111
- Support for AMODE 31- and 64-bit callers with data above or below the bar.
- Support for UTF-8, UTF-16 (big endian only), IBM-1047, IBM-037, and several other encodings. See "Encoding support" on page 49 for more information.
- XML processing features
  - Parsing with schema validation (validation with DTD not supported)
  - Parsing document fragments, see "Parsing XML document fragments with validation" on page 17
  - Parsing of Extensible Dynamic Binary XML (XDBX) streams with validation, see "Parsing XDBX input streams" on page 5
  - Support in the parsed data stream for offsets back into the original source document
  - Optionally return fully qualified element names in end element records
  - Support for XML 1.0 (fourth edition) and XML 1.1 (second edtion).
  - Newline normalization, see "EBCDIC encoding considerations" on page 50
  - Attribute value normalization
  - Omit or return comments in the parsed data stream
  - Optionally return significant white space in unique white space records (instead of character data records)
  - Support for namespaces in XML 1.0 (second edition) and XML 1.1, see "Namespace declarations" on page 46
  - Entity resolution, see "Resolving entity references" on page 46
  - Partial DTD processing, see "Processing DTDs" on page 46

  - Dynamic discovery of schema location information, see "Obtaining information on schema locations" on page 19
  - Restrict the root element name, see "Restricting the root element name" on page 16
  - Support for obtaining additional error information on a validating parse, see "Obtaining additional error information" on page 20.
  - Support to continue a non-validating parse when an undefined prefix is encountered.
- User exits for system services, see Chapter 8, "z/OS XML System Services exit interface," on page 145
- Query service for determining document characteristics, see Chapter 3, "Querying XML documents," on page 9
- Diagnostic support (Chapter 9, "Diagnosis and problem determination," on page 155), including:
  - Diagnostic area, see "Diagnostic Area" on page 157
  - Slip trap support, see "SLIP trap for return codes from the z/OS XML parser" on page 158
  - ARR recovery routine, see "ARR recovery routine" on page 158
  - IPCS formatting, see "XMLDATA IPCS subcommand" on page 155
- Segmented input and output (the entire document does not have to reside in a single buffer), see "Spanning buffers" on page 44
- Mapping macro interfaces for parsed data
- "Enable offload to specialty engines" on page 8

z/OS XML System Services functions

z/OS XML System Services include the following three primary functions:

- A query service that allows callers to determine the encoding of the document and acquire information from the XML declaration.
- Parsing with schema validation
- Parsing without validation

These functions are provided in the form of callable services. A caller can access these services through the z/OS XML System Services APIs (for information on the APIs, see "z/OS XML XL C/C++ API" on page 54 and Chapter 7, "z/OS XML parser API: Assembler," on page 111). The following two sections provide a summary of the functions, with pointers on where to go for more information.

Querying XML documents

XML documents have characteristics that affect the way they are parsed, and the kinds of information that the parser generates during the parse process. One such characteristic is the encoding scheme of the document, which the z/OS XML parser must know before parsing. Using the query service will allow the caller to acquire this information, after which it can then pass it to the z/OS XML parser. The z/OS XML parser will then be able to use the correct encoding scheme to parse the document. For more on this service, see Chapter 3, "Querying XML documents," on page 9.

Parsing XML documents without validation

The non-validating parse process consists of three fundamental steps: initialize the parser, parse the document, and terminate the parser. Multiple documents may be parsed using either a single instance of the parser, or several distinct instances as the caller requires. For more information on this procedure and the individual services called, see Chapter 4, “Parsing XML documents,” on page 11.

Parsing XML documents with validation

Parsing with validation follows the same basic process as for a non-validating parse, with a couple of differences. Firstly, the validating parser must be loaded into storage prior to use. Secondly, an additional step is required to load a pre-processed version of the schema used to validate the document during the parse. This binary schema, referred to as an Optimized Schema Representation (OSR) can be loaded once, and used to validate any document that conforms to it.

For more information on loading the validating parser, see “Loading the validating parser code” on page 13. For more information on OSRs, see “Optimized Schema Representation” on page 8.

Parsing XML document fragments with validation

The validating parser provides support for parsing XML document fragments. The W3C XML specification allows parsing of such document fragments as external parsed entities. The document fragment can be the value of a single attribute, as well as a single element and its descendants. It may be followed by comments and processing instructions. The parser must be provided ancestor and namespace context information to ensure proper validation.

The following example illustrates the usefulness of validating parser support for parsing XML document fragments. Consider a large XML document representing an employee list of the form:

```
<root>
   <Person>
      <name mgr = "NO">Bill</name>
      <age>60</age>
   </Person>
   ....
   <Person>
      <name mgr = "NO">Joe</name>
      <age>45</age>
   </Person>
</root>
```

Each `Person` element is a fragment. If the caller wants to add another `Person` element fragment and validate that it adheres to an associated schema, with validating fragment support, the caller can perform a validating parse of just the individual fragment to be added and then insert the fragment into the document. Prior to this support, the caller would have had to perform a validating parse of the entire document after inserting the new fragment.

For more information on parsing document fragments, see “Parsing XML document fragments with validation” on page 17.

Note: The non-validating parser does not support fragment parsing.

Parsing XDBX input streams

z/OS XML supports the parsing of Extensible Dynamic Binary XML (XDBX) streams with validation. This allows a caller to pass binary XML streams, which are more compact and which can be processed by the validating parser using fewer resources. The result is a conventional z/OS XML record stream. See http://www.ibm.com/support/docview.wss?&amp;uid=swg27019354 for more information on the XDBX binary XML format.

For more information on parsing XDBX input streams, see “Parsing XDBX input streams” on page 47.

Document processing model

There are three main components required for parsing XML documents: input buffer, z/OS XML parser, and output buffer. These three components and their interrelationships make up the processing model. There may be more than one input and output buffer, depending on the size of the document being parsed. If the document is sufficiently large, the caller may find it necessary to provide it to the parser in several pieces, using buffer spanning to maintain the document structure as it is being parsed. Similarly, the caller may need to provide multiple buffers to contain the data stream that the z/OS XML parser generates. For more information on how buffer spanning works, see “Spanning buffers” on page 44.

Document processing is the creation of the output buffers from the parsed input data. As the z/OS XML parser traverses through the input buffer, the output buffer is built. See “Parsed data model” on page 24 for more information on this format.

The following is a diagram of the processing model using buffer spanning. It shows both the input and output buffers, where buffers 2-5 represent the additional buffers created to support a large document.

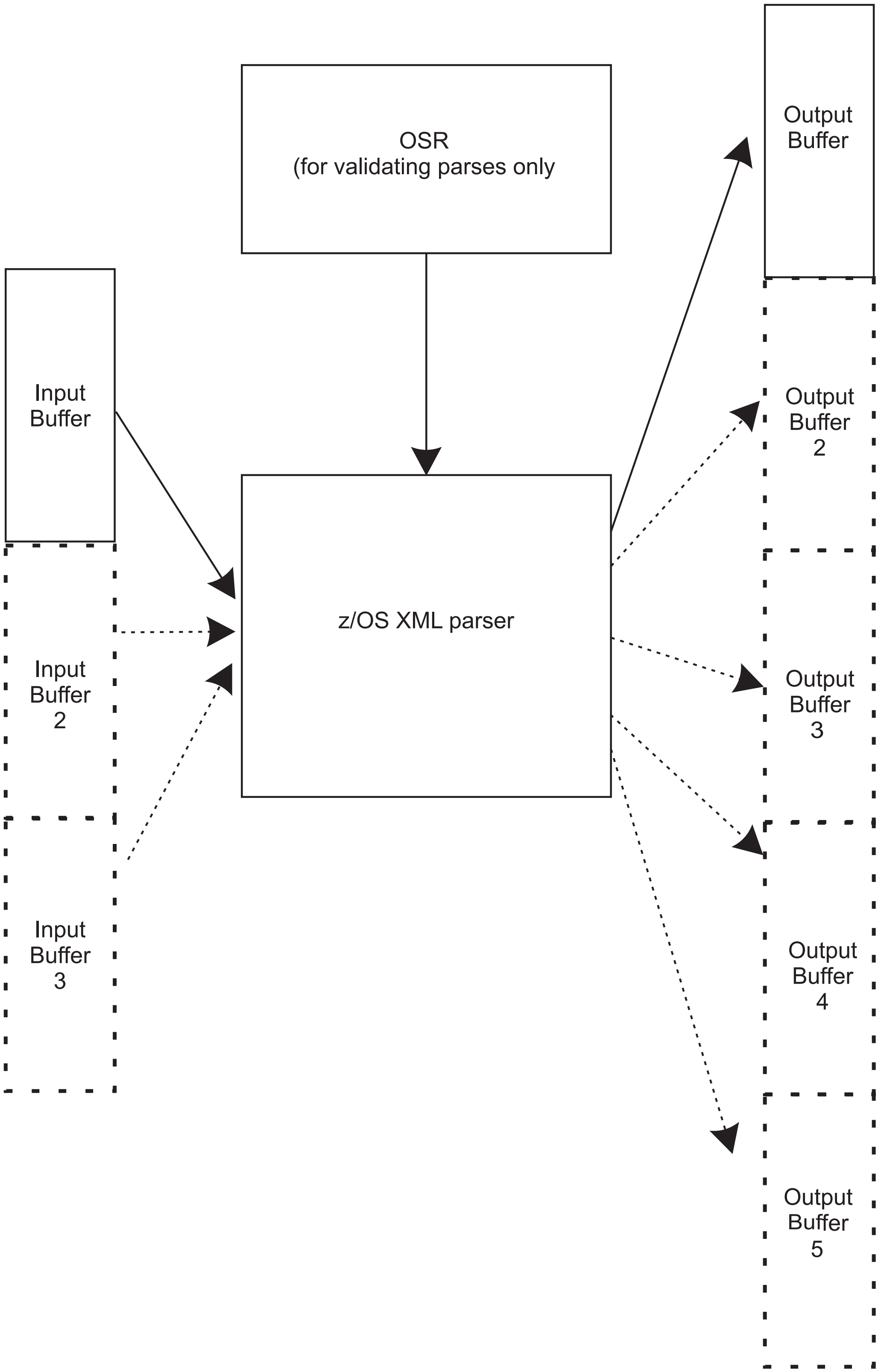

Figure 1. Processing model

For more on how to parse XML documents using the z/OS XML parser, see Chapter 4, “Parsing XML documents,” on page 11.

Output buffer format

The output buffer contains the parsed data stream that results from the parse process. This data stream will contain the parsed XML document contents, along with header and any error information that was produced during the parse. For more information on the format of the output data stream, see "Parsed data model" on page 24.

Optimized Schema Representation

Optimized Schema Representations (OSRs) are pre-processed versions of schemas. They are more easily and more efficiently handled than schemas in text form. When parsing with validation, this form of schema is utilized. An OSR API is provided to assist in the generation, loading and manipulation of these specialized schemas. For information on how to use OSRs, see "Using Optimized Schema Representations" on page 15. For more information on the OSR API, see "OSR generator API" on page 83. For information on performing a validating parse, see Chapter 4, "Parsing XML documents," on page 11.

String Identifiers

String Identifiers (StringIDs) are a special type of output data used to represent some of the strings that the z/OS XML parser encounters during a parse. A StringID is a 4 byte numeric value used to represent a complete string of text, thereby substantially reducing the size of the parsed data stream for documents containing frequently recurring strings, like namespace references. StringIDs can only be used if the optional StringID exit service is activated. For more information on StringIDs, see "String Identifiers" on page 38.

Memory management

The z/OS XML parser provides a memory allocation/deallocation exit allowing callers to provide a pair of allocation/deallocation services. For callers that do not provide a memory allocation exit, the z/OS XML parser provides an option at initialization time allowing the caller to specify how the z/OS XML parser's default routine allocates memory. For more information on these services and the special initialization time feature, see "Managing memory resources" on page 50.

Enable offload to specialty engines

z/OS XML System Services provides the ability for parsing operations to be run on specialty processors: an IBM® System z® Application Assist Processor (zAAP) or an IBM System z10™ Integrated Information Processor (zIIP). The z/OS XML parser, when running in TCB mode, is eligible to run on a zAAP, in environments in which one or more zAAPs are configured. The z/OS XML parser, when running in enclave SRB mode, is eligible to run on a zIIP processor, in environments where one or more zIIPs are configured. Ancillary z/OS XML System Services, such as the query service and the control service, as well as the StringID exit and memory management exits, are not eligible to run on specialty processors. Running of z/OS XML System Services parsing operations on a specialty processor occurs transparently to the calling application.

Chapter 3. Querying XML documents

About this task

An XML document contains declarations that may need special handling during a parse. For instance, if the encoding of the document to parse is unknown, the query service provided by z/OS XML parser can be used to help determine the encoding in order to provide the correct Coded Character Set IDentifier (CCSID) to the parser, when the actual parse is performed.

In order for the caller to query an XML document, all the caller needs to do is use the query service (gxlpQuery for C/C++ callers, GXL1QXD (GXL4QXD) for assembler callers). This service allows the caller to obtain all the XML characteristics of the document. These characteristics can be either the default values or those explicitly contained in an XML declaration. Once these characteristics are obtained, the caller can then determine the encoding scheme needed to parse the document, along with any additional steps that may be needed.

For example, if the document in question uses an encoding scheme of UTF-16, it will require that the z/OS XML parser also uses the UTF-16 encoding scheme when parsing this document. The caller would use the query service to ascertain the encoding type of the document being parsed. Once this information is acquired, the z/OS XML parser can be initialized using the initialization service (gxlpInit for C/C++ callers, GXL1INI (GXL4INI) for assember callers, see Chapter 4, "Parsing XML documents," on page 11) passing the encoding scheme to parse the UTF-16 encoded document.

Note: The query service is the only service that provides support for both UTF-16 (little endian) and UTF-16 (big endian), whereas the other services only support UTF-16 (big endian).

The CCSID value returned by the query service can be used to invoke Unicode Services in order to convert the input document into one of the encodings supported by the z/OS XML parser.

For more information on the query service, see "gxlpQuery — query an XML document" on page 80 for C/C++ callers, and "GXL1QXD (GXL4QXD) — query an XML document" on page 138 for assembler callers. For more information on document encoding support, see "Encoding support" on page 49.

Header files and data macros

This section provides information on the various header files and data macros associated with the z/OS XML parser query service. The names and purposes of these files are listed below:

Note: For each item below, the name of the header file is listed first, followed by the name of the corresponding assembler macro (if any).

gxlhxec.h, GXLYXEC
Contains assorted constant values that are used in the parsed data stream,

values used for assorted fields of the API, and minimum sizes for data areas passed to the z/OS XML parser.

gxlhqxd.h, GXLYQXD
Maps the data area returned from the query service.

gxlhxr.h, GXLYXR
Contains mnemonic values that describe the return and reason codes generated by the z/OS XML parser.

For information on these header files and data macros, see Appendix D, “C/C++ header files and assembler macros,” on page 215.

Chapter 4. Parsing XML documents

Before the z/OS XML parser can perform a parse on an XML document, it must first establish a context in which it can operate. This is accomplished when the caller invokes the initialization routine and passes in a piece of memory where the z/OS XML parser establishes a Parse Instance Memory Area (PIMA). This is the area where the z/OS XML parser creates a base for the internal data structures it uses to complete the parse process.

Rule: A particular PIMA must only be used during the parse of a single XML document at a time. Only after the parse is complete and the parse instance is reset can a PIMA be reused for the parse of another document.

In addition to control information, the PIMA is used as a memory area to store temporary data required during the parse. When the z/OS XML parser needs more storage than was provided in the PIMA, additional storage is allocated. Because allocating additional storage is an expensive operation, the PIMA should be initially allocated with sufficient storage to handle the expected document size, in order to optimize memory allocation requests.

Rule: For the non-validating z/OS XML parser, the minimum size for the PIMA is 128 kilobytes. For the validating z/OS XML parser, the minimum size for the PIMA is 768 kilobytes.

Everything that the z/OS XML parser needs to complete the parse of a document is kept in the PIMA, along with any associated memory extensions that the parser may allocate during the parse process. The caller also must provide input and output buffers on each call to the parse service (gxlpParse for C/C++ callers, GXL1PRS (GXL4PRS) for assember callers). In the event that either the text in the input buffer is consumed or the parsed data stream fills the output buffer, the z/OS XML parser will return XRC_WARNING, along with a reason code indicating which buffer (possibly both) needs the caller's attention. It also indicates the current location and number of bytes remaining in each buffer by updating the *buffer_addr* and *buffer_bytes_left* parameters passed in on the parse request (for C/C++ callers, see the description of "gxlpParse — parse a buffer of XML text" on page 77; for assembler callers, see the description of "GXL1PRS (GXL4PRS) — parse a buffer of XML text" on page 135). This process is referred to as buffer spanning. For more information, see "Spanning buffers" on page 44.

If the entire document has been processed when the z/OS XML parser returns to the caller, the parse is complete and the caller proceeds accordingly. If the caller requires another document to be parsed, it has the option of terminating the current parse instance by calling the termination service (gxlpTerminate for C/C++ callers, GXL1TRM(GXL4TRM) for assembler callers). This will free up any resources that the z/OS XML parser may have acquired and resets the data structures in the PIMA. If the caller needs to parse another document, it will have to call the initialization service again to either completely re-initialize an existing PIMA that has been terminated or initialize a new PIMA from scratch.

Another option is to use the finish/reset function of the z/OS XML parser control service (gxlpControl for C/C++ callers, GXL1CTL (GXL4CTL) for assembler callers) to reset the PIMA so that it can be reused. This is a lighter-weight operation that preserves certain information that can be reused across parsing operations for multiple documents. This potentially improves the performance for subsequent parses, since this information can be reused instead of rebuilt from scratch. Reusing the PIMA in this way is particularly beneficial to callers that need to handle multiple documents that use the same symbols (for example, namespaces and local names for elements and attributes). The PIMA can only be reused in this way when the XML documents are in the same encoding.

Restriction: The following restrictions apply when conducting a validating parse:

- When parsing in non-Unicode encodings, non-representable character entities are replaced with the "-" character prior to validation. See “Non-representable characters” on page 46 for more information on non-representable character entities.
- There is a maximum of 64 KB non-wildcard attributes for a single element, and 64 KB elements in an `All` group.

Steps for parsing XML documents without validation

About this task

The following steps summarize the process of parsing XML documents using the z/OS XML parser:

Procedure

1. Call the initialization service. This establishes the PIMA, which is then used to create and store the initial data structures required to begin the parse process.
2. Call the parse service to parse the document.

   Note: During the parse process and before the end of the document is reached, if the input buffer is empty or the output buffer is full, a warning is issued and the parse service is stopped. Otherwise, the parse service will continue until the document is fully processed.
3. The application processes the output buffer.
4. Determine if there are additional documents to be processed. If so, call the termination service to terminate the existing parse process, and repeat Steps 1-3.

   Tip: For increased performance, the caller can use the control service in place of the termination and initialization services. The control service enables the PIMA to be reused, avoiding the need to free resources and initialize a new PIMA. However, the PIMA can only be reused in this way when the XML documents are in the same encoding. See “gxlpControl — perform a parser control function” on page 54 and “GXL1CTL (GXL4CTL) — perform a parser control function” on page 114 for more information on the control service.

Support for undefined namespace prefix toleration

The default behavior when the non-validating parser encounters an undefined prefix on an element or attribute is to report a XRSN_NS_ELEM_PREFIX_NOT_DECL or XRSN_NS_ATTR_PREFIX_NOT_DECL reason code and terminate the parse. In the event that the caller does not want this error to terminate the parse, they may use the XEC_CTL_ERROR_HANDLING control option to override this default behavior and continue parsing. The “undefined prefix:local name” will then be returned as the local name field in the output buffer. The XEH_Error_Tolerated bit in the XEH_Flags field will be set in the record header when this occurs.

The XEC_CTL_ERROR_HANDLING control option will enable this feature on a control call by way of the XERR structure which is mapped by GXLHERR in the gxlhctl.h file. The XERR_TOL_UNDECL_NS_PREFIX flag in the XERR_ERROR_TOLERATION field in the XERR structure (when set) will cause the parser to continue parsing when this condition occurs.

In addition to continuing on this error, an auxiliary information record may be generated in the output buffer if desired. This will contain the tolerated return and reason codes and the error offset. In order for this record to be returned the XERR_ERROR_INFORMATION flag must be set in the XERR_FLAGS field. This requires the XERR_TOL_UNDECL_NS_PREFIX flag also be set. This new record will have a type of XEC_TOK_AUX_INFO (0xF0FF) and an AUX type of XEC_TOLERATED_ERROR (0x0110). For the format of this record, see “Aux info record - TOLERATED_ERROR” on page 36.

Introduction to data types

There are several data types that can be returned in the output buffer. Therefore, the caller must know what type of data is being returned to effectively process it. The following topics discuss the various data types:

- “Parsed data model” on page 24 - overview of the structures that make up the data stream produced by the parser.
- “Length/Value pairs” on page 37 - the default representation of strings that have been parsed from the original XML document.
- “String Identifiers” on page 38 - a unique numeric value returned by the z/OS XML parser that represents a given text string (a StringID exit service must be provided by the caller to generate these IDs).
- “Metadata records” on page 26 - data records that contain metadata about the parse stream or error information

Loading the validating parser code

Prior to parsing XML documents with validation, the validating parser code must be loaded into storage. To do this, add the validating module, GXLIMODV, to the link pack area, which will make it available to all programs on the system. The size of the GXLIMODV module is ≥ 3 megabytes. Adding this module to LPA will reduce the size of the private area in every address space by this amount. If you have applications that do not use the validating parser that are already storage constrained, then the LPA approach may not be acceptable to your installation. For the exact size of GXLIMODV, run the AMBLIST utility.

If GXLIMODV is not installed into the link pack area, the application must load it into storage. In the non-CICS environment, the application can use the GXL*LOD APIs to load the validating parser into private storage. The load API should be done once per application instance, making validating parser available for use by the entire application. For more information on the load APIs, see “gxlpLoad — load a z/OS XML function” on page 76.

In the CICS® environment, if GXLIMODV is not in the link pack area, it can be loaded into the CICS private region by running the program list table (PLT) program, GXLINPLT at CICS start time or as a transaction program. For more information on setting up and running the CICS PLT program see “Setting up and running the CICS PLT program” on page 22. Loading GXLIMODV into CICS private will take up approximately 3 megabytes of private storage in each CICS region where the PLT program is run. If you are running many CICS regions on the same system, consider using the LPA approach to reduce real storage usage or paging. For size information of module GXLIMODV, which is installed in SYS1.SIEALNKE, use a utility such as AMBLIST.

Steps for parsing XML documents with validation

About this task

The following steps summarize the process of parsing XML documents using the z/OS XML parser with validation:

Procedure

1. Call the parser load service or run the XML CICS PLT program if in the CICS environment. This will load the parser into storage. For more information, see "Loading the validating parser code" on page 13.
2. Call the OSR initialization utility. This establishes the OSR generator Instance Memory Area (OIMA), which is then used as the work area for the OSR generator.
3. Call the OSR generator utility. This utility creates an OSR from one or more text-based schemas passed to the OSR generator instance, using the load schema utility.

   Note: An OSR can be saved and then used for parsing future documents that share the same schema(s) from which the OSR was generated. As a result, steps 2 and 3 may not be required each time an XML document is parsed using validation.
4. Call the parser initialization service. This establishes the PIMA, which is then used to create and store the initial data structures required to begin the parse process.
5. Call the control service. This will load the generated OSR into the z/OS XML parser.
6. Call the parse service to parse the document.

   Note: During the parse process and before the end of the document is reached, if the input buffer is empty or the output buffer is full, a warning is issued and the parse service is stopped. Otherwise, the parse service will continue until the document is fully processed.
7. The application processes the output buffer.
8. Determine if additional schemas need to be processed. If so, repeat steps 3, 5 and 6. If you want to reuse an existing OSR, use the OSR load utility.
9. Determine if there are additional documents to be processed. If so, call the termination service to terminate the existing parse process, and repeat steps 1 -7.

   Tip: For increased performance, the caller can use the control service in place of the termination and initialization services. The control service enables the PIMA to be reused, avoiding the need to free resources and re-initiate a new PIMA. However, the PIMA can only be reused in this way when the XML documents are in the same encoding. See "gxlpControl — perform a parser control function" on page 54 and "GXL1CTL (GXL4CTL) — perform a parser control function" on page 114 for more information on the control service.

Using Optimized Schema Representations

Optimized Schema Representations (OSRs) are specialized forms of schemas used during the validating parse process. They can be created from utilities provided by the OSR generator API. For more information about the OSR generator API, see "OSR generator API" on page 83.

Setting up the environment

About this task

Before the caller can begin generating OSRs, some environment variables must be set. The following lists the environment variables that must be set along with their appropriate values.

Note:

1. The caller should use the proper 31/64-bit versions of the binaries listed below. Mixing versions of different binaries will result in unpredictable results.
2. The OSR generator is only supported with IBM 31-bit SDK for z/OS, Java 2 Technology Edition, V5 and above.

LIBPATH
must include paths to the following:

- For C API callers only (gxlcosr1.dll for 31-bit, gxlcosr4.dll for 64-bit) - /usr/lib
- For 31-bit callers - /usr/lib/java_runtime
- For 64-bit callers - /usr/lib/java_runtime64
- Java binaries and JVM
  - For 31-bit callers -
    - /usr/lpp/java/J5.0/bin
    - /usr/lpp/java/J5.0/bin/j9vm
  - For 64-bit callers -
    - /usr/lpp/java/J5.0_64/bin
    - /usr/lpp/java/J5.0_64/bin/j9vm

CLASSPATH
must include paths to the following:

- The Java API callers only (gxljapi.jar) - /usr/include/java_classes

  Note:

  1. Do not include gxljosrgimpl.jar. It will be loaded from /usr/include/java_classes
  2. Callers of the Java API must choose the 31- or 64-bit version of Java that they intend to use. They may either specify the explicit path to the required executable (/usr/lpp/java/J5.0/bin/java for 31-bit, /usr/lpp/java/J5.0_64/bin/java for 64-bit), or include the path to the required Java version in their PATH variable. Users of the C API and command interfaces do not need to be concerned with this.

Usage tips

Tips are provided below to facilitate the usage of OSRs:

- An OSR is not a schema library. In other words, you should not throw all necessary schemas into a single OSR and use it similar to a library.

- Schemas should reference one another by way of the `<xs:import ...>` construct. That is, OSRs are meant to contain hierarchies of schemas, where one or more schemas reference others to handle increasingly more specific structures in the source XML document being transformed.
- You should consider creating one schema OSR to validate entire classes of documents.

The OSR used for validation becomes part of the parse instance, and remains in use for all validating parse requests until a different one is specified through the control service. Callers who use buffer spanning to pass documents to and from the parser in pieces should know that schemas cannot be changed in the middle of the parse process. A control request to specify a different schema will cause a reset of the parse instance so that the next parse request must be for a new XML document.

Note: For callers using schemas written in XML 1.1 format, use IBM Java Technology Edition V6.

Restricting the root element name

While performing a validating parse, the caller has the option to restrict the root element name to a list of one or more root name or namespace pairs. When selecting this option, validation is performed on the root name in the document being parsed. This option is only available for validating parses.

To enable this option, during the parse step the caller must perform a control call (gxlpControl) with the control option GXLHXEC_CTL_RESTRICT_ROOT prior to parsing the document to indicate that the root name is to be validated. The caller must also pass along a data area in the format of GXLHXRR, which contains the list of root names. Failing to do this will cause an error, resulting in the z/OS XML parser needing to be reset using CTL_FIN. The root names are specified by a local name (root name) and an optional URI for the root namespace. The strings passed in to the control call (gxlpControl) call must be in the encoding of the z/OS XML parser configured at initialization time.

The control call (gxlpControl) prepares the z/OS XML parser for a new document, but the current feature set is preserved. Subsequent resets (such as CTL_FEAT) will not change the current settings of the restrict root element control call. These settings will still apply when parsing subsequent documents.

The information produced in the output buffer from the subsequent parse does not change when using this option.

The caller can remove the restriction on the root element name by calling gxlpControl () with the control option GXLHXEC_CTL_RESTRICT_ROOT and setting the XRR_ENTRY_COUNT value to '0' in the GXLHXRR data area.

The following is an example call sequence:

```
gxlpLoad
gxlpInit
gxlpControl(GXLHXEC_CTL_LOAD_OSR)
gxlpControl(GXLHXEC_CTL_RESTRICT_ROOT,GXLHXRR)
gxlpParse
gxlpTerminate
```

Parsing XML document fragments with validation

Before beginning document fragment parsing, the caller must specify the fragment path. The namespace binding information is also required when there is namespace context associated with the fragment path. To load the fragment context, the caller needs to issue a gxlpControl call with the control option XEC_CTL_LOAD_FRAG_CONTEXT. This must occur before document fragment parsing is enabled .

Note: A new OSR with the extra Fragment Parsing Table information is required in order to parse a document fragment with validation. Pre-z/OS V1R12 OSRs cannot be used in this parsing environment. To load the OSR, the caller needs to issue a control call with the control option XEC_CTL_LOAD_OSR. This must occur before document fragment parsing is enabled.

To enable fragment parsing, the caller needs to issue a gxlpControl call with the control option XEC_CTL_FRAGMENT_PARSE and set the XFP_FLAGS_FRAGMENT_MODE bit to 'ON' in the control data structure. This must occur prior to issuing the gxlpParse call with the XML document fragment loaded into the input buffer. The z/OS XML parser will perform regular parsing including well-formedness checking and validation, however the root element is not required. The XML declaration and Doctype Declaration are not allowed as part of the document fragment.

If the z/OS XML parser reaches the end of the input buffer, and the parsed document fragment is well-formed, the z/OS XML parser ends the parse successfully. If the z/OS XML parser reaches the end of the input buffer, and the parsed document fragment is not well-formed, the z/OS XML parser will return an error. Otherwise, the z/OS XML parser will return to request more input or output buffer space. Fragment parsing with validation is restricted to a single attribute, as well as a single element and its descendants, optionally followed by comments and processing instructions. Attempts to parse multiple element fragments with validation will result in an error . If the caller decides to finish parsing the document fragment, and the z/OS XML parser returns to request for more input and output buffer space during fragment parsing, an error will occur. A gxlpControl call with the control option XEC_CTL_FIN must be issued in order to parse another document or document fragment.

When the caller finishes parsing a document fragment, they must issue a gxlpControl call with the control option XEC_CTL_FRAGMENT_PARSE and set the XFP_FLAGS_FRAGMENT_MODE bit to 'OFF' in the control data structure to notify the z/OS XML parser that fragment parsing has been disabled.

The following is an example call sequence for a validating fragment parse:

```
gxlpLoad
gxlpInit
gxlpControl(XEC_CTL_LOAD_OSR)
gxlpControl(XEC_CTL_LOAD_FRAG_CONTEXT, FPATH)
gxlpControl(XEC_CTL_FRAGMENT_PARSE) -- enable the fragment mode
gxlpParse
gxlpControl(XEC_CTL_FRAGMENT_PARSE) -- disable the fragment mode
gxlpTerminate
```

If the caller wants to perform document fragment parsing or non-fragment parsing on a different document, a gxlpControl call with the control option XEC_CTL_FIN must be issued prior to a gxlpParse call. This XEC_CTL_FIN operation will reset and prepare the current parse instance for a new document parse. The loaded fragment context will remain in storage and become active when fragment mode is enabled again. If the next document fragment to be parsed requires different fragment path or namespace binding information, then a new XEC_CTL_LOAD_FRAG_CONTEXT control call must be made to update this information. Failure to load the correct information may cause unexpected results such as well-formedness or validation errors.

The following is an example call sequence for a validating parse with fragments from two different documents:

```
gxlpLoad
gxlpInit
gxlpControl(XEC_CTL_LOAD_OSR)
gxlpParse
gxlpControl(XEC_CTL_LOAD_FRAG_CONTEXT, FPATH1)
gxlpControl(XEC_CTL_FRAGMENT_PARSE, BIT:ON) -- enable fragment mode #1
gxlpParse -- parse document fragment #1 part1 in FPATH1
gxlpParse -- parse document fragment #1 part2 in FPATH1
gxlpControl(XEC_CTL_FRAGMENT_PARSE, BIT:OFF) -- disable fragment mode #1
gxlpControl(XEC_CTL_FIN)
gxlpControl(XEC_CTL_LOAD_FRAG_CONTEXT, FPATH2)
gxlpControl(XEC_CTL_FRAGMENT_PARSE, BIT:ON) -- enable fragment mode #2
gxlpParse -- parse document fragment #2 in FPATH2
gxlpControl(XEC_CTL_FRAGMENT_PARSE, BIT:OFF) -- disable fragment mode #2
gxlpTerminate
```

Restrictions: Validation in fragment parsing cannot satisfy all aspects of schema validation for an arbitrary input string as various aspects of schema validation refer to other aspects of the document. Although it may be possible to validate some aspects of the following schema constructs, in general they require the entire document to be available. The restriction falls into two broad categories: those things that cannot be validated reasonably and those things that can be validated in isolation but could possibly fail within the context of a document. The first category is avoided by requiring the client to ensure that the name of the element matches the element that it is replacing. The second category includes the following:

Namespaces

The gxlpControl API allows the establishment of a namespace context in which to do the validation. If none are provided, it will be assumed that there are no namespaces bound to prefixes other than those bound within the input to be validated.

ID/IDREF

This requires knowledge of other portions of the instance document and will only be validated using the appropriate simple content validator.

Unique elements and attributes

Unique elements and attributes are contained within the subtree of the element containing the unique schema indicator. If this element is the root for the document fragment or is a descendant, then the unique element or attribute can be handled normally. However, if the unique specifier is an ancestor of the root, there could be collisions which will not be detected.

KEY/KEYREF

This is similar to ID/IDREF in that the KEYs must be unique and KEYREFs must match a KEY, and similar to unique attributes in that it must be contained in the subtree. Those aspects that can be checked are validated, but those aspects that refer to ancestral content are not validated.

Validating attributes and elements with attributes (other than xsi:type)
These attributes are meaningless in fragments and are only validated using the appropriate simple type validator. This means that the special characteristics of these attributes will not have an effect. For example, schemaLocation will not indicate a schema but will just be validated against string.

xsi:type
When xsi:type is an attribute on an element, it will have the expected effect. Validating this as a single attribute will result in it being validated using the qname simple type validator.

DTDs Because the z/OS XML parser is only passed the document fragment, it has no knowledge of the entity definitions or default attribute values in the internal DTD subset. Therefore, if the schema contains an attribute with a type of ENTITY, it will fail. It will also fail if it relies on a default attribute value defined in the internal DTD subset.

Comments, processing Instructions and annotations
These constructs can be included in the normal ways within an element being validated.

Attributes that rely on an xsi:type attribute to also be present
When validating an attribute, the attribute is validated using the type containing the attribute. Therefore, it cannot be a derived attribute or an attribute only available on a derived type.

The parser cannot determine the actual particle when an element is indicated
The parser cannot determine the actual particle when an element is indicated. Rather, the designator is used to indicate a type so information on the particle, such as fixed values and `nillable` are not checked.

The impact of the validation within its context is not checked
The impact of the validation within its context is not checked. Therefore, the effects of changing an element to a different element (likewise with attributes) are not checked. Checking such characteristics requires validating the parent and document fragment does not provide this information.

Obtaining information on schema locations

When parsing a document containing schema references, the caller generates and loads an OSR. In order to make sure that the appropriate OSR is loaded, the caller must determine which schemas are referenced in the document and their locations. To this end, the caller can query the XML document for namespaces and schema locations.

The caller can obtain information on the schema location by initializing the PIMA with the GXLHXEC_FEAT_SCHEMA_DISCOVERY feature. This will cause the z/OS XML parser to pause after parsing the start tag of the root element. The output buffer is then populated with records as if a normal parse was performed, with the following additional records: GXLHXEC_TOK_ROOT_ELEMENT and GXLHXEC_TOK_SCHEMA_LOCATION. GXLHXEC_TOK_ROOT_ELEMENT contains the root element name and GXLHXEC_TOK_SCHEMA_LOCATION contains the schema location information. The output buffer will not contain any start element, attribute value, or namespace declaration records. After the end of the start tag has been reached and all schema info records have been outputted, the z/OS XML parser provides the caller an opportunity to load an OSR before the parse is continued. If the parse is continued, whichever OSR was loaded by a

GXLHXEC_CTL_LOAD_OSR operation will be used to validate the document. If no OSR loading operation has been performed since parser initialization, the parser must be reset in order to parse again.

The following is an example sequence, with the GXLHXEC_FEAT_SCHEMA_DISCOVERY enabled:

```
gxlpLoad
gxlpInit (with GXLHXEC_FEAT_SCHEMA_DISCOVERY enabled)
gxlpParse
gxlpControl(GXLHXEC_CTL_LOAD_OSR)
gxlpParse
gxlpTerminate
```

If the document does not contain either a schemaLocation or noNamespaceSchemaLocation attribute, then GXLHXEC_TOK_SCHEMA_LOCATION records will not appear in the output stream .

See "gxlpInit — initialize the z/OS XML parser" on page 72 and "GXL1INI (GXL4INI) — initialize a parse instance" on page 131 for more information on using this feature.

Obtaining additional error information

The default behavior when there is an XML document error is the z/OS XML parser returns a reason code which identifies the error, and an offset into the original document which is being parsed. In many cases, this is insufficient as XML documents are commonly transcoded in transit. Such transcodings can cause offsets to change in the document, rendering the offsets less useful.

The type of error information that can be provided will necessarily vary by the particular error encountered. The following are examples of additional error information that may be made available when an error is encountered:

- A bit indicator which identifies the general location within the document. This can be XML declaration, DTD, element, miscellaneous, or an attribute which is a namespace declaration. If the error occurs in entity replacement text, an additional bit will be set.
- A location string which represents a path-like expression for the ancestor element node(s) at the point of the error. This will be only applicable when the error occurs within the XPath addressable portion of the document. There also will be an associated namespace context provided in order to assist in correctly identifying the failing location.
- A failed string which represents information taken from the document. For example, this could be an incorrect element name or an invalid Unicode character.
- A string which is expected to be present but is missing.

Because element occurrences can be repeated with the same names, it is possible to also include position information in the path-like expression. However, tracking element positions will be detrimental to performance even when there is no error.

This additional error information may be obtained for a validating parse by using the XEC_CTL_ERROR_HANDLING control option along with the XERR structure which is mapped by GXLHERR in gxlhctl.h to enable this feature. The XERR_ENH_ERROR_INFORMATION flag will cause additional auxiliary records to be returned which may contain information on the location of the error, the string which is in error and also possibly an expected string. The XERR_XD_PTR is where the service will store the address of the diagnostic area, which is mapped by GXLHXD in the gxlhxd.h file. The XD_LastOutput field is a pointer to the data area containing these records. This data area is within the PIMA and is formatted in the same manner as a normal output buffer. It will have a buffer information record followed by one or more additional records. This data area will be overlaid on a subsequent call to the z/OS XML parser.

The XERR_ENH_ERROR_LOCATION feature flag may also be specified to request that position indexes be returned in the XPath expression which represents the error location. Enabling this feature will impact performance.

In order to explain how element position indexes are tracked, the concept of an expanded QName needs to be explained. See the following example:

```
<?xml version="1.0"?>
<root>
<pre:elem1 xmlns:pre="http://w3.pok.ibm.com">
</pre:elem1>
<pfx:elem1 xmlns:pre="http://w3.pok.ibm.com">
</pfx:elem1>
</root>
```

For the first non-root element, the QName would be `pre:elem1` and the second `pfx:elem1`. However, an expanded QName consists of the namespace URI and the local name. Two expanded QNames are equal if their namespace URIs and local names are equal (even if the prefixes are not equal). So in this case, if an error occurred on the `pfx:elem1`, the index would be “[2]” since both expanded QNames are `http://w3.pok.ibm.com:elem1`. This is a simple example. With default namespaces, the situation can be more complicated.

XML Path language

XML Path (XPath) is a language for addressing parts of an XML document. It is a W3C recommendation. XPath is well known and commonly used in XML applications. This language will be used for specifying location path expressions which covers most areas of an XML document.

The following are considered nodes in the XPath language:

- Document root
- Elements
- Attributes that are not namespace declarations
- Processing instructions (PIs)
- Comments
- Text

During the progress of a particular parse, constructs will become “XPath identifiable” when sufficient characters are parsed to uniquely recognize the type of node and its name (if it has one). The following are the points where this occurs for each node:

Document root
This is identifiable when the first non-XML declaration structure is discovered. This will never have a corresponding name.

Element
This is identifiable after the element type is fully parsed. This will be when either whitespace, a '>' character or a '/' character are encountered after the element type.

Attribute
This is identifiable after the attribute name is fully parsed. This will be when either whitespace or an '=' character are encountered after the attribute name.

PI This is identifiable after the PI target is fully parsed. This will be when either whitespace or a “?” character are encountered after the PI target.

Comment
This is identifiable after the beginning of the comment markup is parsed. This will be after the “” are encountered.

Text This is identifiable after the “>” of markup within element nodes.

Some structures within an XML document are not identifiable using the XPath language. The following constructs are not XPath identifiable:

XML declaration
The path location string will be 0 length. This is recognized when “<?xml “ is encountered at the beginning of the document and before the subsequent “>" is encountered.

Doctype declaration
The path location string will be 0 length. This is recognized when "<?DOCTYPE " is encountered and before the subsequent “]”.

Namespace declarations
The path location string will denote the containing element node. This is recognized when “xmlns” is encountered where there should be an attribute.

While text nodes are XPath identifiable, they will not be uniquely denoted. Instead, the location path will denote the containing parent element node.

For more information on the format of the auxiliary information records, see “Metadata records” on page 26.

Setting up and running the CICS PLT program

About this task

The GXLINPLT assembler program uses the MVS™ LOAD command on the z/OS XML System Services module, GXLIMODV, the validating parser. GXLINPLT is distributed as part of z/OS and is installed in `SYS1.SIEALNKE(GXLINPLT)`. By default, it is in the LNKLST.

The following steps are required to add GXLINPLT as a program list table (PLT) program in CICS and to run at CICS start up:

Procedure

1. Define GXLINPLT to the CICS CSD. See Appendix H, “CICS examples,” on page 237for an example of a job that uses the DFHCSDUP program to define GXLINPLT to the CICS CSD. See CICS Resource Definition Guide for more information on DFHCSDUP programs. See CICS Resource Definition Guide for more information on the CICS CSD.
2. Add the CICS group that contains the GXLINPLT program in a GRPLST that is included at CICS startup. Here is an example: add a new group, GXLXMLCG to GRPLST GXLXMLCL, and add GXLXMLCL to the GRPLST parameter in the DFH$SIPx file.

3. Customize the Program Load Table (PLT) to include the z/OS XML System Services program, GXLINPLT, to run during the second stage of initialization. For an example of a job to update the PLT table, see Appendix H, “CICS examples,” on page 237. For this example, the DFH$SIPx would include the entry: PLTPI=I1. Next, add the load module where the program (in the above example, the DFHPLTI1 program) is installed, to the CICS DFHRPL concatenation.
4. Add the data set where GXLINPLT is installed to the CICS DFHRPL concatenation. By default, this data set is `SYS1.SIEALNKE`.

   Note: GXLINPLT may also run as a CICS transaction.

Results

For more information on how to setup and run CICS PLT programs, see the CICS Customization Guide.

Header files and data macros

This section provides information on the header files and data macros associated with the z/OS XML parser. The names and purposes of these files are listed below:

Note: For each item below, the name of the header file is listed first, followed by the name of the corresponding assembler macro (if any).

gxlhxml.h

Contains prototypes for all of the API entry points, as well as include statements for all of the other header files that are required for the API. The Metal C version of this header also includes logic to call either the 31 or 64 bit version of the requested API, depending on the addressing mode of the caller.

There is no corresponding assembler version of this header file.

gxlhxeh.h, GXLYXEH

Describes all of the structures that the z/OS XML parser generates in the parsed data stream. This includes both the records that represent the individual markup and content parsed from the document, as well as metadata about the data stream itself.

gxlhxec.h, GXLYXEC

Contains assorted constant values that are used in the parsed data stream, values used for assorted fields of the API, and minimum sizes for data areas passed to the z/OS XML parser.

gxlhqxd.h, GXLYQXD

Contains the structure that describes the information returned from the Query XML Declaration (QXD) service. It also contains constants that enumerate the allowable values for certain fields of the structure.

gxlhxd.h, GXLYXD

Maps the z/OS XML parser extended diagnostic area.

gxlhxr.h, GXLYXR

Contains mnemonic values that describe the return and reason codes generated by the z/OS XML parser.

gxlhxsv.h, GXLYXSV
Maps the system service vector that the caller uses to describe the exits that it provides to the z/OS XML parser.

gxlhctl.h, GXLYCTL
Contains the various structures that are used in the gxlpControl (GXL1CTL/GXL4CTL) service.

gxlhxft.h, GXLYXFT
Maps the input and output area used by the control feature flag of the gxlpControl GXL1CTL (GXL4CTL) service.

gxlhxosr.h, GXLYXOSR
Maps the input and output area used by the optimized schema representation.

For information on these header files and corresponding data macros, see Appendix D, "C/C++ header files and assembler macros," on page 215.

Parsed data model

This section provides information on the data model used to represent the contents of the output buffer. The caller needs to understand this data model so that it can effectively process the parsed data stream that has been created in the output buffer.

The z/OS XML parser produces a structured data stream resulting from the parse process. It is a feature that distinguishes the z/OS XML parser from most other XML parsers. The parsed data stream consists of a set of self-describing records representing the output of the parser. These records provide a structure to the data stream that allows a consumer to navigate the data stream as needed. Some of the records represent the actual semantic content of the parsed document, while others provide metadata about the parse itself. There may be more than one group of these records (or record groups) in a single output buffer. This can occur if the input buffer spans multiple times before the output buffer is filled.

Common record header

Each record in the parsed data stream consists of a common header, followed by information that is specific to a given record type. The common header has the following structure:

Table 1. Common record header

	Fields		
+0	record type (2 bytes)	flags (1 byte)	reserved (1 byte)
+4	record length		

The record type determines the form of the data that immediately follows the header and which makes up the body of the record. The record flags provide information about the specific record to which they belong. Each bit of the flags byte has the following meaning:

Table 2. Record flag bits

Bit position	Name	Purpose
0	XEH_CONTINUED	This record is continued in the next output buffer.

Table 2. Record flag bits (continued)

Bit position	Name	Purpose
1	XEH_NO_ESCAPES	There are no characters that need to be escaped in this record.
2	XEH_DEFAULT	This record is supplied with default content from a DTD or a schema.
3	XEH_ERROR_TOLERATED	This record contains an undefined prefix which is tolerated because this behavior was requested.

The XEH_No_Escapes flag is provided as an aid to callers that need to re-serialize the parsed data stream back to an XML document in text form. It is relevant only for records that represent character data or attribute values (its meaning is undefined for all other records). It indicates that there are no special characters present that need to be escaped in the text of the record during re-serialization. The set of these special characters is made up of "<", ">", "&", and the single and double quotes. The caller must substitute either one of the well known strings ("&lt;", "&gt;", "amp;", "&apos;", "&quot;") or a numeric character reference in the serialized text in order to create a well formed XML document.

Note: Single and double quotes are allowed within character data. If they appear within character data, they are not considered escaped characters. However, if single or double quotes exist within attribute values, they are considered escaped characters.

When this flag is on, the caller can safely avoid scanning the text associated with the record to look for characters that must be escaped during re-serialization. When the flag is off, one of the special characters may be present, and such a scan is required. Note that there are certain instances involving buffer spanning when it is not possible for the parser to determine that this bit should be set. As a result, for character data and attribute value records that span multiple output buffers, the XEH_No_Escapes bit may be off, even when there are actually no characters that need to be replaced during serialization. If the bit is on though, it will always be safe to avoid scanning for characters that need escaping.

The flags field is followed by 1 reserved byte and the record length. The record length contains the total length of the record - including the header. Navigating from one record to the next is done by moving a pointer, by the specified record length, from the first byte of the current record header.

Record (token) types

Record types are values used to identify the purpose of each record parsed from the input document. The record type, along with the data stream options in the buffer info record (see “Buffer info record” on page 26), indicates the form of the record. Record forms are a means of indicating the number of values that make up the record itself, and are described in a separate section below. Here are the record types returned by the z/OS XML parser (their definitions are provided in gxlhxec.h for C and C++ callers, and GXLYXEC for assembler callers):

Table 3. Record types

Token name	Meaning
GXLHXEC_TOK_BUFFER_INFO	information about the buffer containing the parsed data stream

Table 3. Record types (continued)

Token name	Meaning
GXLHXEC_TOK_ERROR	error information
GXLHXEC_TOK_XML_DECL	an XML declaration
GXLHXEC_TOK_START_ELEM	start of an element
GXLHXEC_TOK_END_ELEM	end of an element
GXLHXEC_TOK_ATTR_NAME	name of an attribute
GXLHXEC_TOK_ATTR_VALUE	value of an attribute
GXLHXEC_TOK_NS_DECL	a namespace declaration
GXLHXEC_TOK_CHAR_DATA	character data
GXLHXEC_TOK_START_CDATA	start of a CDATA section
GXLHXEC_TOK_END_CDATA	end of a CDATA section
GXLHXEC_TOK_WHITESPACE	a string of white space characters
GXLHXEC_TOK_PI	processing instruction
GXLHXEC_TOK_COMMENT	a comment
GXLHXEC_TOK_DTD_DATA	DOCTYPE declaration information
GXLHXEC_TOK_UNRESOLVED_REF	an entity reference that cannot be resolved
GXLHXEC_TOK_AUX_INFO	auxiliary information about individual items in the parsed data stream
GXLHXEC_TOK_SCHEMA_LOCATION	schema location information
GXLHXEC_TOK_ROOT_ELEMENT	root element name

The above token names are for the C/C++ callers. Assembler callers use token names without the prefix "GXLH".

Most of the record types listed above fall into one of four classes, based on the number of values they contain from the document being parsed. Two of these record types - the buffer info and error records - are different (see “Buffer info record”and “Error info record” on page 28) because they contain metadata about the information in one of the buffers (input or output), rather than data parsed from the input stream. The form of the data they contain is unique to the purpose of the record.

The data structures that describe this data stream can be found in the data model header file gxlhxeh.h for C/C++ callers, and the mapping macro GXLYXEH for assembler callers. Data is not aligned on any kind of boundary, and there are no alignment requirements for the input or output buffers provided by the caller.

Metadata records

Some records contain metadata related to the parsing process. These records are discussed below.

Buffer info record

Because the data stream that the z/OS XML parser generates in the output buffer consists of one or more groups of records, each group always begins with the buffer info record - a record containing metadata about the parsed data stream contained in the current output buffer. This record includes the length of the buffer used by the record group and flags indicating the characteristics of the data stream.

The following is the structure for the buffer info record, including the record header:

Table 4. Buffer info record structure

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	datastream options		
+C	parse status	reserved	
+10	buffer length used		
+14			
+18	offset to error record		
+20			

This record is not allowed to span output buffers, so the continuation flag in the record flags field of the buffer header will always be zero. The datastream options contain a flag indicating whether or not StringIDs are in use, plus some of the flags from the feature flags parameter on the z/OS XML parser init call. These flags indicate some characteristic of the data in the parsed data stream. The full list of flags indicate:

- StringIDs are in effect
- Comments are stripped (GXLHXEC_FEAT_STRIP_COMMENTS)
- White space is being tokenized (GXLHXEC_FEAT_TOKENIZE_WHITESPACE)
- Returning CDATA as CHARDATA (GXLHXEC_FEAT_CDATA_AS_CHARDATA)
- Validating parser is enabled (GXLHXEC_FEAT_VALIDATE)
- Source offsets are enabled (GXLHXEC_FEAT_SOURCE_OFFSETS)
- Full end tag feature is enabled (GXLHXEC_FEAT_FULL_END)

Note:

1. The GXLHXEC_FEAT* flags in above parentheses are defined in gxlhxec.h for C/C++ callers and GXLYXEC for assembler callers. For assembler callers, remove the "GXLH" prefix from the constant names.
2. The buffer info record is mapped out in gxlhxeh.h for C/C++ callers and GXLYXEH for assembler callers.

The "parser status" field is another set of flags in the buffer info record. If an unresolved external reference is present in this buffer, the unresolved reference bit will be on. If a non-representable character reference is present in this or a subsequent buffer for this document, the non-representable character reference bit will be on.

The "buffer length used" field indicates the portion of the output buffer consumed by the group of records represented by this buffer info record. If no buffers are spanned during the parse process, there will be only one buffer info record present in the output buffer, representing a single group of records. If buffers are spanned, there may be several record groups, each with corresponding buffer info records present in the output buffer. The number of record groups and buffer info records depends on how the caller manages the buffers that are passed to the parser. See “Spanning buffers” on page 44 for more information.

The "error record offset" field indicates the offset from the beginning of the buffer info record to the beginning of the error info record. If this offset is zero, there is no error record present in the group of records represented by the buffer info record.

Error info record

The error info record is placed in the parsed data stream whenever a parsing error is detected. The offset to the error from the start of the document, along with the return and reason code generated by the z/OS XML parser when the error was encountered, are kept in a field of the error info record. Here is the structure of the record, including the record header:

Table 5. Error info record structure

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	return code		
+C	reason code		
+10	offset of the error from the start of the document		
+14			

Note: The error info record is mapped out in gxlhxeh.h for C/C++ callers, and GXLYXEH for assembler callers.

For information on error codes and how to use them, see “Using return and reason codes” on page 51.

Aux info record

When the source document offsets feature (GXLHXEC_FEAT_SOURCE_OFFSETS) is selected, or the character reference record (GXLHXEC_UNREPRESENTABLE_CHARREF_REC) is requested, an information record is inserted in the output buffer. The record has the following structure:

Table 6. Aux info record

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	aux flags	information type	
+C	-varied information-		

The record values are defined as follows:

Record header
This is the standard record header of all records in the data model. The record type is GXLHXEC_TOK_AUX_INFO.

Aux flags
These flags provide information about the form of the data in the rest of the record:

- XEH_AUX_LONG_VALUE - This flag is only used in records which contain values which can vary in size. This bit will be OFF if the record contains integer values that are 4 bytes in length. The bit will be ON if the record contains values that are 8 bytes in length. Any value or record length fields in the record such as the record length in the header will always be 4 bytes no matter what the value of this bit is.

  All offset values which are under 4GB-1 in magnitude will be represented as a 4 byte value in the data stream and the XEH_AUX_LONG_VALUE flag will be OFF. When an offset is encountered which exceeds 4 GB, then all offset records from that point on will be represented as 8 byte values in the data stream and the XEH_AUX_LONG_VALUE will be set ON.
- AUX_ENTITY - This is set for information records that are generated from entities.

Information type
This value identifies what information is contained in the record. See information types for details on the different types.

-varied information-
The contents of the additional information will depend on the information type and flags. For offset records, this will contain either a 4 or an 8 byte value which represents the offset of the particular structure from the beginning of the document. It will be 4 bytes if the XEH_AUX_LONG_VALUE bit flag bit is OFF in the header. It will be 8 bytes if the XEH_AUX_LONG_VALUE bit flag is ON in the header.

Information types:

GXLHXEC_OFFSET_START_STARTTAG
This is the offset of the '<' at the beginning of an XML start tag. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_START_ELEM .

GXLHXEC_OFFSET_END_STARTTAG
This is the offset of the '>' at the end of an XML start tag. This record occurs in the datastream immediately after the last GXLHXEC_OFFSET_END_ATTRVALUE, if there are attributes, or the GXLHXEC_OFFSET_END_STARTTAGNAME record if there are no attributes.

GXLHXEC_OFFSET_END_STARTTAGNAME
This is the offset to the end of the XML start name qname. This record occurs in the datastream immediately following the GXLHXEC_TOK_START_ELEM.

GXLHXEC_OFFSET_START_ATTRVALUE
This is the offset of the beginning quote of the attribute value. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_ATTR_VALUE.

GXLHXEC_OFFSET_END_ATTRVALUE
This is the offset of the ending quote of the attribute value. This record occurs in the datastream immediately after the GXLHXEC_TOK_ATTR_VALUE.

GXLHXEC_OFFSET_START_COMMENT
This is the offset of the '<' at the beginning of an XML comment. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_COMMENT.

GXLHXEC_OFFSET_END_COMMENT
This is the offset of the '>' at the end of an XML comment. This record occurs in the datastream immediately following the GXLHXEC_TOK_COMMENT.

GXLHXEC_OFFSET_START_CDATA
This is the offset of the '<' at the beginning of an XML CDATA. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_START_CDATA.

GXLHXEC_OFFSET_END_CDATA
This is the offset of the '>' at the end of an XML CDATA. This record occurs in the datastream immediately following the GXLHXEC_TOK_END_CDATA.

GXLHXEC_OFFSET_START_PI
This is the offset of the '<' at the beginning of an XML PI. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_PI.

GXLHXEC_OFFSET_END_PI
This is the offset of the '>' at the end of an XML PI. This record occurs in the datastream immediately following the GXLHXEC_TOK_PI.

GXLHXEC_OFFSET_START_XMLDECL
This is the offset of the '<' at the beginning of an XML Declaration. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_XML_DECL.

GXLHXEC_OFFSET_END_XMLDECL
This is the offset of the '>' at the end of an XML Declaration. This record occurs in the datastream immediately following the GXLHXEC_TOK_XML_DECL.

GXLHXEC_OFFSET_START_ENDTAG
This is the offset of the '<' at the beginning of an XML end tag. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_END_ELEM.

GXLHXEC_OFFSET_END_ENDTAG
This is the offset of the '>' at the end of an XML end tag. This record occurs in the datastream immediately following the GXLHXEC_TOK_END_ELEM.

GXLHXEC_OFFSET_START_DTD
This is the offset of the '<' at the beginning of an XML DOCTYPE declaration. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_DTD_DATA.

GXLHXEC_OFFSET_END_DTD
This is the offset of the '>' at the end of an XML DOCTYPE declaration. This record occurs in the datastream immediately following the GXLHXEC_TOK_DTD_DATA.

GXLHXEC_OFFSET_START_NSVALUE
This is the offset of the quote at the beginning of an XML namespace declaration value. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_NS_DECL.

GXLHXEC_OFFSET_END_NSVALUE

This is the offset of the quote at the end of an XML namespace declaration value. This record occurs in the datastream immediately following the GXLHXEC_TOK_NS_DECL.

GXLHXEC_OFFSET_ROOT_ELEMENT

This is the offset of the < at the beginning of the root element start tag. This record occurs in the datastream immediately preceding the GXLHXEC_TOK_ROOT_ELEMENT.

GXLHXEC_CHARREF_UNREP_REC

This record type contains information about non representable character references in the document.

Note: This information type contains a different auxiliary information record than the previous information types. The variable section of the record is as follows:

Table 7. Alternate structure for variable section of aux info record (GXLHXEH_AUX_LONG_VALUE = OFF)

	Fields
+0	The binary value of the character reference that cannot be represented.
+4	The offset of the character reference that cannot be represented in the document.
+8	The offset into the string of the previous replacement character's record in the output record.

If the GXLHXEH_AUX_LONG_VALUE bit is set to 'ON' in the GXLHXEH_AUX flag, the variable section of the record has the following structure:

Table 8. Alternate structure for variable section of aux info record (GXLHXEH_AUX_LONG_VALUE = ON)

	Fields
+0	The binary value of the character reference that cannot be represented.
+4	-reserved-
+8	The offset of the character reference that cannot be represented in the document.
+C	
+10	The offset into the string of the previous replacement character's record in the output record.
+14	

The above information type names are for the C/C++ callers. Assembler callers use information type names without the "GXLH" prefix.

Entities and default XML structures

If the records are inserted in the output stream via XML entity replacement or default generation, then offset information records will be generated, and the varied information field will represent the offset of the ';' character of the entity reference in the main document or the '>' character of the element which contains the default attribute. Also, all information records generated from entities will have the entity flag bit set ON.

Default XML structures include any of the following:

Attributes

These can be generated from DTDs or schemas.

Namespace declarations

These can be generated from DTDs or schemas.

Start tags and end tags

These can be generated from schemas only.

Content

These can be generated from schemas only and only within default start and end tags.

Interactions with other features

The source offsets feature can interact with other features. The following is a list of those features, along with an explanation of the interaction:

Strip comments (GXLHXEC_FEAT_STRIP_COMMENTS)

When source offsets are enabled, comment records will continue to be stripped. However, the source offset information records for comment markup will continue to be inserted into the output.

CDATA as char data (GXLHXEC_FEAT_CDATA_AS_CHARDATA)

When source offsets are enabled, CDATA will continue to be outputted as character records. However, the source offset information records for CDATA markup will continue to be inserted into the output. In this case, the order of the information records will not be in document order in relation to the data in the character records.

Validation

Information records will be created when using the validating parser as well as the non-validating parsing.

Fragment parsing (GXLHXEC_CTL_FRAGMENT_PARSE)

Normally, source offset values are presented based on the beginning of the XML document, However, when document fragment parsing is enabled, the source offset values are presented based on the fragment parsing block constituted by the start and end fragment parsing control call. Therefore, the source offset will start from zero for every document fragment that is parsed after the start fragment parsing control call is made.

Aux info record - Error_Location

This is a type of aux info record. This record pertains to the XEC_CTL_ERROR_HANDLING features. It has the following format:

Table 9. Aux info record - Error_Location

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	aux flag	aux type = 0x0101	
+C	error flags		
+10	error location path length		
+14	error location path value		
+n	error namespace context length		
+n+4	error namespace context value		

- The record header flags field will always be 0.
- The record length field contains the total length of the record - including the header
- The XEH_AUXFlag field will always be 0.
- The aux record type field will have the value 0x0101 - XEC_ERROR_LOCATION

The flags field (XEH_ErrFlags) consists of the following possible values:

XEH_ERRXMLDECL
Error occurred in the XML declaration or text declaration portion of the document.

XEH_ERRDTD
Error occurred in the document type declaration portion of the document.

XEH_ERRELEMENT
Error occurred in the element portion of the document.

XEH_ERRMISC
Error occurred within PIs, comments in the prolog, miscellaneous areas of the document, or between markup.

XEH_ERRREPTEXT
Error occurred within entity replacement text while resolving an entity reference. This bit will be on in addition to the other bits.

XEH_ERRNSDECL
Error occurs within an attribute which is a namespace declaration.

The error location path field is a string in the form of a length/value pair which represents the approximate location of the failure.

The namespace context is a list of namespace URIs delimited by '/' characters with each step corresponding to the same step in the error location path. Each namespace URI will be surrounded by double quote characters to aid in parsing the string.

Error location path and namespace context

This is defined as a list of element and attribute nodes in the format of an XPath expression which denotes the closest ancestor to the failure point which has passed sufficient well-formedness or validation checks to be XPath identifiable as a node. If the enhanced error information feature is enabled, then position indexes will be included in the XPath expression for any nodes whose position is greater than 1.

Note: In the XPath specification, the position is 1 based.

If the XML document includes namespace definitions, then the nodes in the expression will be namespace prefix qualified as they were in the original XML document. In addition, the namespace context at the point of the failure is provided. If no namespaces URIs are applicable at the point of the failure, then the length of the namespace context field will be 0. If the path denotes a PI or comment, then the corresponding step in the namespace definition will not be present.

If fragment parsing is in progress when the error occurs, the path will only include nodes that had been parsed in the current document fragment. It will not include nodes which were only passed in by way of the load fragment context CTL call.

The error location path follows the definitions in the XPath 1.0 specification. The following XPath constructs are used:

'/' This identifies the root node of the document and includes prolog and miscellaneous nodes which are XPath identifiable as comments and processing instructions. If an error occurs in the root element's type (qname), then the path will be /. If the error location path is present, it will always begin with '/'.

location step
Consists of a node test. The following node tests are supported:

QName
This corresponds to element nodes in the path and will be a prefix:localname if the qname is namespace qualified. Elements using default namespaces will not have a prefix.

@Qname
This corresponds to attribute nodes in the path will be a prefix:localname if the qname is namespace qualified. This can only appear at the end of the path.

comment()
This is used when an error occurs in an XPath identifiable comment.

processing-instruction('name')
This is used when an error occurs in a processing instruction. The 'name' is a string which represents the name of the processing instruction if it was correctly specified in the document.

[n] This is the 1 based position index which will be appended to a qname if the enhanced location feature is enabled and the position index of the node is > 1.

Additional usage notes are as follows:

- Location steps are separated by a '/' character.
- If an error occurs in the XML declaration or DOCTYPE, then the error location length is 0.
- If an error occurs outside the root element with non-XPath identifiable markup or between markup, then the error location path will be a '/'.
- If an error occurs within a namespace declaration, then the location path will denote the parent element node.
- If an error occurs within the an attribute and is not XPath identifiable as an attribute node, the location path will denote the parent element node.
- If an error occurs in a text node, then the location path will denote the parent element node.
- If the path or namespace context would be longer than 2 gigabytes, then they are truncated to 2 gigabytes.

Error location and fragment parsing

There are some special considerations when fragment parsing is enabled:

- If the fragment context denotes that the fragment is an attribute value, then the error location will be a zero length string.
- If the error occurs before or after the main element in the fragment, the error location will be a zero length string.

- The XEH_ErrFlags field can only have the XEH_ERRXMLDECL bit on if the error occurs in the text declaration portion of the fragment, or the XEH_ERRELEMENT bit on if it occurs anywhere else in the fragment.

Aux info record - ERROR_STRING

This is a type of aux info record. This record pertains to the XEC_CTL_ERROR_HANDLING features. It has the following format:

Table 10. Aux info record - Error_String

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	aux flag	aux type = 0x0102	
+C	failing string length		
+10	failing string value		

- The record header flags field will always be zero.
- The record length field contains the total length of the record - including the header
- The XEH_AUXFlag field will always be zero.
- The aux record type field will have the value 0x0102 - XEC_ERROR_STRING

The failing string contains a string in the form of a length/value pair from the document which is associated with the failure. The parser will test the failing byte sequence. If it is in an US-ASCII displayable range of characters, then the character itself will be present in the string. If it is not displayable, it will instead be the hex representation. These will follow the C convention of 0xnn. For example, if a character is found which is not allowed in an xml document, then it may show here as 0xC270.

In cases where the XEH_ERRREPTEXT bit is on in the error location record, this string will contain the entity reference in the main document which led to the error occurrence.

Aux info record - EXPECTED_STRING

This is a type of aux info record. This record pertains to the XEC_CTL_ERROR_HANDLING features. It has the following format:

Table 11. Aux info record - Expected_String

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	aux flag	aux type = 0x0103	
+C	expected string length		
+10	expected string value		

- The record header flags field will always be zero.
- The record length field contains the total length of the record - including the header
- The XEH_AUXFlag field will always be zero.
- The aux record type field will have the value 0x0103 - XEC_EXPECTED_STRING

The expected string contains a string in the form of a length/value pair which shows a string which was expected in the document in order for the document to parse correctly.

If there is more than one option for what is required at any point, this record will not be present.

Aux info record - TOLERATED_ERROR

This is a type of aux info record. This record pertains to the XEC_CTL_ERROR_HANDLING features. The information data has the following format:

Table 12. Aux info record - TOLERATED_ERROR

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	aux flag	aux type = 0x0110	
+C	error return code		
+10	error reason code		
+14	error offset		
+18			

- The record header flags field will always be zero.
- The record length field contains the total length of the record - including the header
- The XEH_AUXFlag field will always be zero.
- The aux record type field will have the value 0x0110 - XEC_TOLERATED_ERROR.
- Error return code, error reason code and error offset depend on the errors.
- The error offset will be a 64-bit field.
- In the event that source offset auxiliary records are also being returned, this record will immediately follow those records for the element or attribute in the output buffer.

Extended end element record

If the XEC_FEAT_FULL_END feature is enabled, then the XEC_TOK_END_ELEM record will be generated as a Record Form 3 instead of Record Form 0. Here are the contents of the record when StringIDs are disabled:

Table 13. Extended end element record (no StringID)

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	length of Lname		
+C	value of Lname		
+10			
+14	length of URI		

Table 13. Extended end element record (no StringID) (continued)

	Fields
+18	value of URI
+1C	
+20	length of prefix
+24	value of prefix
+28	

Here are the contents of the record when StringIDs are enabled:

Table 14. Extended end element record (StringID)

	Fields		
+0	record type	flags	reserved
+4	record length		
+8	StringID of Lname		
+C	StringID of URI		
+10	StringID of prefix		

Default content flag (XEH_DEFAULT)

When an output record is generated from a definition in the DTD or schema, the XEH_DEFAULT flag bit will be set in the record header flags field. This bit will indicate that an attribute, namespace or element was generated from the DTD or schema.

31- and 64-bit compatibility

The length and offset fields outlined in the metadata records above are all 64-bit values, with associated 31-bit versions to provide 31- and 64-bit compatibility. Assembler callers in 64-bit mode can pass in buffer lengths greater than 2 GB to GXL4PRS. As a result, the z/OS XML parser may have values in length and offset fields that are much greater than 2 GB. 31-bit assembler callers are limited to 2 GB, and should reference the XEH_*31 fields in order to use the proper value. The XEH_*31 fields are in GXLYXEH . These fields can also be found in gxlhxeh.h for C/C++ callers.

Note: The offset of the error from the start of the document, when the input document is segmented and the sum of the segment sizes is greater than 2 GB, may be a 64-bit value even though the caller may only be 31-bit.

Length/Value pairs

Strings that have been parsed from the original XML document (qualified name components, character data, comment text, etc.) are, by default, represented by length/value pairs. This length indicates the actual length of the text represented by the pair. There are no string terminators, such as a NULL character used to indicate the end of a piece of text. Length fields may be zero, indicating that a particular string is not present (for example, the namespace string length for an element that is not namespace qualified will be zero), and the value length will also be zero. In the absence of a String Identifier exit (see “String Identifiers” on page 38), all strings in the parsed data stream are represented by a length/value pair.

String Identifiers

This section provides information on the String Identifiers (StringIDs) that can be passed back to the caller by the z/OS XML parser.

Note: The StringID exit is an optional service that the caller may supply. If there is no StringID exit available, the z/OS XML parser will simply return the actual length/value pairs for the strings representing localnames, URIs, and prefixes in the data stream it returns to the caller. See “Length/Value pairs” on page 37for more discussion on this topic.

StringIDs are 4 byte numeric values that are used to represent a given string that is returned from the z/OS XML parser to the caller. StringIDs can be used to represent the localname (lname), namespace prefix, and namespace URI for the following items:

- element names
- attribute names
- namespace declarations

These are the strings in the parsed data stream that are most likely to be repeated. StringIDs are provided by a caller-supplied service exit that the z/OS XML parser invokes any time it encounters certain strings that it hasn't seen before. See the description of the symbol service exit (“GXLSYM31 (GXLSYM64) — StringID service” on page 151, “GXLPSYM31 (GXLPSYM64) — StringID handler” on page 108) for more details.

Once the z/OS XML parser receives a StringID for a given string, it will record the ID, and return it in place of the actual lname, namespace prefix, or namespace URI string in the parsed data stream that is returned to the caller. The use of StringIDs reduces the size of the parsed data stream especially for documents with namespace references. URIs that would normally be returned for every element and attribute name can be represented in 4 bytes instead of their text that is generally much longer.

Record forms

The general form of a record created in the parsed data stream contains a fixed header section, followed by zero or more values. These values may consist of either a length and value pair, or a single StringID value, depending on the type of data being represented, and the data stream options that are in use. StringIDs are used to represent attribute and element name components - the lname, namespace URI, and namespace prefix for start element and attribute name records, and the namespace prefix and URI for namespace declarations. When StringIDs are not in use, these name components are represented by length and value pairs, just like other types of data returned in the records that make up the parsed data stream.

Each record begins with a fixed section that contains the record type, a set of flags, and the length of the entire record. This is followed by the values relevant to the specific type of information represented by the record. In most cases, these values represent an individual item parsed from the XML document. The exceptions are the metadata records (the buffer info and error records), which contain information describing the input and output streams, but which are not directly related to a specific item from the XML document.

The record length field is the value that must be used to navigate from one record to the next in the parsed data stream. Although the lengths and types of the individual fields of a record are explained below, the caller must not use these to calculate the location of subsequent records.

The data stream options contained in the buffer info record of each output buffer, and the token types of each record within those buffers uniquely identify the type of information contained in each record. This type information is reflected in the record form used for each record. These structures are defined in the header file "gxlhxeh.h (GXLYXEH) - mapping of the output buffer record" on page 215. For assembler callers, they are defined in GXLYXEH. Also, see Table 21 on page 41 for a description of the various record types.

Record form 0

This is a simple record that is used to describe items in the output stream that have no associated value. It consists of only a record header.

Table 15. Record form 0

Fields		
record type	flags	reserved
record length		

Record form 1

These records describe items in the output stream that have one associated value - most often a character string.

Table 16. Record form 1

Fields		
record type	flags	reserved
record length		
value 1 length		
bytes 1 to n of value 1		

These records are used to return things like character data to the caller. StringIDs are never used in these records.

Record form 2

These records describe items in the output stream that contain two values. There are two variations of this record form, depending on whether or not StringIDs are being used. Namespace declaration records are examples of these. In the case where StringIDs are provided by the caller through the GXLSYM31 (GXLSYM64) StringID service exit, the record form looks like the following:

Table 17. Record form 2 (with StringID)

Fields		
record type	flags	reserved
record length		
StringID for value 1		
StringID for value 2		

When StringIDs are not in use, values one and two are represented as conventional length and value pairs:

Table 18. Record form 2 (without StringID)

Fields		
record type	flags	reserved
record length		
value 1 length		
bytes 1 to n of value 1		
value 2 length		
bytes 1 to n of value 2		

There are other form 2 records that will always use length and value pairs, regardless of whether or not StringIDs are available. Processing instructions are an example of this kind of record, since the target and value of a processing instruction are always returned as strings represented by length and value pairs.

Record form 3

Records of this form are for parsed data that is described by 3 separate values. These records include those for element and attribute names, which can contain either StringIDs or length and value pairs, as well as XML declarations, which are always represented by the length and value pair version of this record form. Here is what the StringID based version of this form looks like:

Table 19. Record form 3 (with StringID)

Fields		
record type	flags	reserved
record length		
StringID for value 1		
StringID for value 2		
StringID for value 3		

The following is the length and value pair version of the record form:

Table 20. Record form 3 (without StringID)

Fields		
record type	flags	reserved
record length		
value 1 length		
bytes 1 to n of value 1		
value 2 length		
bytes 1 to n of value 2		
value 3 length		
bytes 1 to n of value 3		

Field values by record type

The following is a complete listing of the descriptions of values for each record type. The actual type of certain values will differ, depending on the use of StringID.

Table 21. Field values by record type

Record type	Record form	Contains StringIDs	Value number	Value description
GXLHXEC_TOK_ATTR_NAME	3	No	1	length and value of Lname
			2	length and value of namespace URI
			3	length and value of namespace prefix
GXLHXEC_TOK_ATTR_NAME	3	Yes	1	StringID of Lname
			2	StringID of namespace URI
			3	StringID of namespace prefix
GXLHXEC_TOK_ATTR_VALUE	1	-	1	length and value of attribute value
GXLHXEC_TOK_AUX_INFO				
GXLHXEC_TOK_BUFFER_INFO	N/A	N/A	-	See "Buffer info record" on page 26
GXLHXEC_TOK_COMMENT	1	-	1	length and value of comment
GXLHXEC_TOK_CHAR_DATA	1	-	1	length and value of character data
GXLHXEC_TOK_DTD_DATA	3	-	1	length and value of root element name
			2	length and value of public identifier
			3	length and value of system identifier
GXLHXEC_TOK_END_CDATA	0	-	-	none
GXLHXEC_TOK_END_ELEM	0	-	-	none

Table 21. Field values by record type (continued)

Record type	Record form	Contains StringIDs	Value number	Value description
GXLHXEC_TOK_END_ELEM (only used when GXLHXEC_FEAT_FULL_END feature is enabled)	3	No	1	length and value of Lname
			2	length and value of namespace URI
			3	length and value of namespace prefix
GXLHXEC_TOK_END_ELEM (only used when GXLHXEC_FEAT_FULL_END feature is enabled)	3	Yes	1	StringID of Lname
			2	StringID of namespace URI
			3	StringID of namespace prefix
GXLHXEC_TOK_ERROR	N/A	N/A	-	See “Error info record” on page 28
GXLHXEC_TOK_NS_DECL	2	No	1	length and value of namespace prefix
			2	length and value of namespace URI
GXLHXEC_TOK_NS_DECL	2	Yes	1	StringID of namespace prefix
			2	StringID of namespace URI
GXLHXEC_TOK_PI	2	-	1	length and value of PI target
			2	length and value of PI text
GXLHXEC_TOK_ROOT_ELEMENT	2	No	1	length and value of namespace
			2	length and value of Lname
GXLHXEC_TOK_ROOT_ELEMENT	2	Yes	1	StringID of namespace
			2	StringID of Lname

Table 21. Field values by record type (continued)

Record type	Record form	Contains StringIDs	Value number	Value description
GXLHXEC_TOK_SCHEMA_LOCATION	2	No	1	length and value of namespace URI
			2	length and value of schema URI
GXLHXEC_TOK_SCHEMA_LOCATION	2	Yes	1	StringID of namespace URI
			2	none
GXLHXEC_TOK_START_CDATA	0	-	-	none
GXLHXEC_TOK_START_ELEM	3	No	1	length and value of Lname
			2	length and value of namespace URI
			3	length and value of namespace prefix
GXLHXEC_TOK_START_ELEM	3	Yes	1	StringID of Lname
			2	StringID of namespace URI
			3	StringID of namespace prefix
GXLHXEC_TOK_UNRESOLVED_REF	1	No	1	length and value of entity name
GXLHXEC_TOK_WHITESPACE	1	-	1	length and value of a white space string
GXLHXEC_TOK_XML_DECL	3	-	1	length and value for version
			2	length and value for encoding
			3	length and value for standalone

The above token names are for the C/C++ callers. Assembler callers use token names without the "GXLH" prefix.

Spanning buffers

The z/OS XML parser is built to handle documents that may be larger than any single buffer the caller can pass to the z/OS XML parser. When buffers need to be spanned (because either the text in the input buffer is consumed, or the parsed data stream fills the output buffer), the z/OS XML parser returns a conditional success return code (XRC_WARNING), and a reason code that indicates which buffer caused the spanning condition. The caller then should handle the spanning buffer, and can optionally manage the other buffer as well.

For example, if the z/OS XML parser indicates that the output buffer is full on a return to the caller after saving and refreshing the output buffer pointers, the caller may choose to refill the input buffer with more text to parse before calling the parse service again to continue the parse process. This will require either moving the unparsed text to the front of the current input buffer, or to a new input buffer, and filling in the remainder with more unparsed text. In this way, the caller potentially reduces the number of times the z/OS XML parser has to return to the caller because of a spanned buffer during the parse of a document.

The z/OS XML parser will advance the input and output pointers to the byte after the last byte that the parser processed in each buffer. Similarly, it will update the *bytes_left* parameters to indicate the number of unprocessed or unused bytes in each buffer. The caller must use the reason code returned from the z/OS XML parser to tell which buffer must be handled and which buffer may optionally be handled. The caller cannot rely on either the *address* values or the *bytes_left* values to tell which buffer has spanned.

Splitting records

When building the parsed data stream in the output buffer, the z/OS XML parser will always ensure that all records are fully formed. Since some records represent items from the document that may be very long (for example, CDATA, white space, or comments), certain types of records are deemed to be splittable. In these cases, the z/OS XML parser will always ensure that the header for the split record is complete, but the value(s) in the record will only contain a part of the item being parsed. A flag in the record header will be set to indicate that the record is continued.

Note: In fragment parsing mode, the flag is set to 'OFF' on a continued character data record when CDATA is outside an element tag (start and end tag). However, if CDATA is inside an element that splits, the continuation flag will still be 'ON'. Split records may span several output buffers if they are very long, or if the output buffers are relatively short.

Records that represent items of fixed length or that contain multiple values are mostly deemed to be non-splittable. If there is no room in the current output buffer to hold them, the entire record will be placed in the next output buffer. These records represent things like start element tags, attribute names, namespace declarations, or end element tags.

Note: The one exception to this rule are processing instructions (PIs). Because the text associated with PIs can be arbitrarily long, they are permitted to split.

If the z/OS XML parser determines that an output buffer is spanned, and requests another buffer to continue processing, the caller needs to return a new buffer large enough to contain a minimum set of complete data. If the item that needs to be placed at the beginning of this new buffer is a non-splittable record that doesn't fit, the z/OS XML parser will return with a return code of XRC_FAILURE, and a reason code of XRSN_BUFFER_OUTBUF_SMALL.

The z/OS XML parser generally does not split records unless there is a need to - for example, to fit into a given output buffer. However, the decision to split a record depends on many factors. There are instances where the z/OS XML parser will split records of the same type within the same buffer, and this is normal. This is particularly true for XDBX streams, where the z/OS XML parser generates records based on the stream of XDBX tags presented by the builder of the stream. One should not expect, for instance, that the stream of z/OS XML records generated for a given text document will have records split in the same way as for an XDBX stream representing the same document.

The following table shows which record types can be split:

Table 22. Splittable record types

Record type	**Splittable?**
GXLHXEC_TOK_ATTR_NAME	No
GXLHXEC_TOK_ATTR_VALUE	Yes
GXLHXEC_TOK_AUX_INFO	No
GXLHXEC_TOK_BUFFER_INFO	No
GXLHXEC_TOK_COMMENT	Yes
GXLHXEC_TOK_CHAR_DATA	Yes
GXLHXEC_TOK_END_CDATA	No
GXLHXEC_TOK_END_ELEM	No
GXLHXEC_TOK_ERROR	No
GXLHXEC_TOK_DTD_DATA	No
GXLHXEC_TOK_NS_DECL	No
GXLHXEC_TOK_PI	Yes
GXLHXEC_TOK_ROOT_ELEMENT	No
GXLHXEC_TOK_SCHEMA_LOCATION	No
GXLHXEC_TOK_START_CDATA	No
GXLHXEC_TOK_START_ELEM	No
GXLHXEC_TOK_UNRESOLVED_REF	No
GXLHXEC_TOK_WHITESPACE	Yes
GXLHXEC_TOK_XML_DECL	No

The above token names are for the C/C++ callers. Assembler callers use token names without the "GXLH" prefix.

Splitting multibyte characters

When a caller segments an input stream for passing to the z/OS XML parser in several parts, the possibility exists that the end of an input buffer falls in the middle of a multibyte character. When this happens, the z/OS XML parser will detect the partial character, and buffer up any bytes for that character from the current buffer before returning to the caller for more input. When the next buffer of input arrives, the z/OS XML parser will virtually prefix the saved bytes of the split character to the beginning of the new buffer, and continue processing. This relieves the caller from having to ensure that multibyte characters at the end of a buffer are complete before calling the z/OS XML parser.

Processing DTDs

z/OS XML System Services will handle internal DTDs for the purpose of processing entity declarations and default attribute value definitions. It only processes entity declarations and default attribute values from the internal DTD. Processing instructions that fall within the internal DTD will be returned to the caller, but no other text from the DTD will be processed. The z/OS XML parser will return a DTD record in the parsed data stream that contains the name of the root element, plus the system and public literals that make up the identifier of any external subset. The content of the internal subset is not returned to the caller.

Resolving entity references

Entities declared in the internal DTD will have all references to them in the root element resolved. These references will have the text from the entity declaration substituted for the reference, and there will be no other indications made in the parsed data stream that an entity reference was present in the parsed document.

Unresolved entities are references to entity names that have no declaration in the internal DTD. Unresolved entities in the root element are tolerated if there is an external subset (standalone="no" in the XML declaration). In this case, if the XEAR_ENTREF_STOP_UNRESOLVED control option is not set, a record of type XEC_TOK_UNRESOLVED_REF is generated in the parsed data stream, with the associated value being the name of the entity. Also in this case, if the XEAR_ENTREF_STOP_UNRESOLVED control option is set, the parse stops and this condition is flagged as an error. When the document only has an internal subset (standalone="yes"), all unresolved entities are flagged as errors.

Non-representable characters

By default, when a character reference which cannot be represented in the current code page is encountered, the z/OS XML parser places a dash (“-“) in the output stream for that character. The caller may use the XEC_CTL_ENTS_AND_REFS control call to specify that a different character appear in the output stream. The caller may also request by way of this control call that an additional output record be placed in the output stream with more information on the character reference.

Namespace declarations

Namespace declaration records are placed in the parsed data stream between the start and end element records for the elements that contain them. This is different than in SAX-like environments where the namespace declaration events precede the start element event for a given element.

Only the namespaces that have been declared within an element, including the default namespace, will have entries in the parsed data stream for that element. The caller may construct the complete namespace context for an element by keeping a stack of namespace declarations as they are encountered in the parsed data stream. Default namespaces will have URI values, but no associated prefix. When a default namespace is unset, it is represented in the parsed data stream as a namespace declaration record with no URI or prefix.

Note: The z/OS XML parser is an XML compliant namespace parser only, and not an XML non-namespace parser. Because of this, if the z/OS XML parser parses a document compliant with the XML non-namespace standard, it can attribute namespace characteristics to an element that is not intended to contain namespaces. This is because non-namespace documents can have a ":" in an element structure that does not actually indicate a namespace. Thus, if non-namespace documents are being parsed, the resulting parsed data stream may not match the expected parsed data stream or the parser may flag the document as erroneous.

Using the z/OS XML parser in a multithreaded environment

The z/OS XML parser can be called from multiple work units (threads/tasks or SRBs) to parse multiple documents at the same time, provided that each parse utilizes a unique Parse Instance Memory Area (PIMA). Multiple work units must not utilize the same PIMA simultaneously, or the z/OS XML parser will behave unpredictably. As long as the caller has a separate PIMA that has been initialized by the z/OS XML parser for each document being processed, multiple documents can be handled simultaneously. A caller may choose to preallocate a pool of PIMAs to be used for parse requests. It is the responsibility of the caller to allocate the PIMA in a subpool that will not be cleaned up while the PIMA is in use. Subpools tied to the job step task are recommended.

Parsing XDBX input streams

Extensible Dynamic Binary XML (XDBX) is a binary XML form composed of both numeric and string data. The numeric data is used for several purposes, including identifying the semantic purpose and length of each associated string in the stream. See http://www.ibm.com/support/docview.wss?&uid=swg27019354 for more information about the format of XDBX streams.

XDBX can be passed to z/OS XML and parsed with validation to create a z/OS XML record stream, in the same way that regular XML text documents are handled. See the appropriate parser API section for details about how to initialize and control a parse instance for XDBX streams. Once the parse instance is initialized and configured, parsing proceeds in the same way as for regular XML text input. Non-validating parse requests are not supported for XDBX streams.

Although the API is called the same way for both XML text and XDBX input streams, there are important differences in the way the parser handles each type of input. More precisely, there is no need for z/OS XML to perform certain low level parsing functions on XDBX streams. Key among these is the need for a low-level scan of the input stream. XDBX streams already have tag fields that describe the meaning of each string and length fields that delimit the strings' boundaries. The z/OS XML parser gains a performance advantage over the validation of XML text input by using the information already provided in the XDBX form.

z/OS XML does not re-scan each string of text from an XDBX stream. Consequently, the no-escapes bit setting is determined entirely from the tag used to represent a given string. This is important for the 'U' (text), 'b' (attribute), and 'W' (whitespace) tags in the XDBX stream. If the XDBX stream creator associated these tags with strings that do in fact contain characters that need to be escaped on serialization of the stream, z/OS XML will not catch this, and will set the XEH_NO_ESCAPES bit in the record header for any associated records generated during validation. Similarly, if a 'T' (text), 'y' (attribute) or a 'C' (CDATA) tag is used when the associated string has no characters that require escaping for serialization, the XEH_NO_ESCAPES flag will be off. This is true even when values are defaulted from the DTD or schema during the validation process.

Another difference from XML text input is that XDBX streams are required to have all entity references resolved. For this reason, none of the z/OS XML functionality implemented for managing unresolved entities is relevant for XDBX input. See the descriptions of the control APIs for more information about how character and entity references are handled for XDBX streams.

Every XDBX stream begins with a magic number (0xCA3B), and is encoded in big-endian form. There is no need for a byte-order-mark, and the parse request will fail if one is present in the XDBX stream.

The following usage notes apply to parsing XDBX streams:

- XDBX input streams may be passed to z/OS XML for parsing with validation when the GXLHXEC_FEAT_XDBX_INPUT feature is enabled. Attempts to initialize a parse instance for an XDBX input stream without validation will result in a failure.
- Validation is performed using an Optimized Schema Representation (OSR) in the same way as for conventional XML text input. The output of the parser is a conventional z/OS XML record stream.
- XDBX input streams contain a combination of binary information and UTF-8 text strings, meaning that the CCSID specified at parser initialization must always be UTF-8.
- Certain other parser features are not currently supported in combination with XDBX streams:
  - GXLHXEC_FEAT_SCHEMA_DISCOVERY
  - GXLHXEC_FEAT_SRC_OFFSETS

  In addition, some control operations are not allowed when the parser is initialized to handle XDBX streams. See the section describing the “gxlpControl — perform a parser control function” on page 54operation for details of those functions that are not compatible with XDBX streams.

Chapter 5. Additional usage considerations

This chapter provides additional usage information for the z/OS XML parser. The following topics are discussed:

- “Recovery considerations”
- “Encoding support”
- “Managing memory resources” on page 50
- “Using return and reason codes” on page 51

Recovery considerations

z/OS XML provides an ARR recovery routine. This recovery routine can be turned on through an initialization option when invoked through the assembler API. For callers of the C/C++ parse API (gxlpParse), when running in Language Environment®, the ARR recovery routine is provided by default in most cases. For more information on the ARR recovery routine, see “ARR recovery routine” on page 158.

Recovery can also be supplied by the caller. Callers who want to clean up z/OS XML parser resources should invoke GXL1TRM (GXL4TRM), the parser termination service, either when the parse completes or if an unexpected error occurs during the parse. The termination service will cause all secondary storage to be freed. It is up to the caller to free the PIMA storage (see “Managing memory resources” on page 50 for more information).

Encoding support

z/OS XML System Services supports several code pages. The caller must supply the CCSID of the encoding for the document at the time the z/OS XML parser is initialized. For a complete listing of the supported code pages, see Appendix I, “Supported encodings,” on page 239. The following table lists more commonly used code pages with their associated CCSID values, along with the equates provided for the caller.

Table 23. Code page CCSID values

Code page	CCSID	Equate Names
UTF-8	1208	GXLHXEC_ENC_UTF_8
UTF-16 (big endian)	1200	GXLHXEC_ENC_UTF_16
EBCDIC/IBM-037	37	GXLHXEC_ENC_IBM_037
EBCDIC/IBM-1047	1047	GXLHXEC_ENC_IBM_1047

Assembler callers use equate names without the "GXLH" prefix.

The query service can be used to query a document's XML declaration so that a caller can determine if the document has to first be converted to one of the supported encodings before parsing begins. This function will return a parsed record for the XML declaration that contains, among other things, a Coded Character Set IDentifier (CCSID) which can be passed to an encoding conversion service, such as Unicode Services, to put the document in a form that the z/OS

XML parser can process. See the description for "gxlpQuery — query an XML document" on page 80 or "GXL1QXD (GXL4QXD) — query an XML document" on page 138 for more information.

EBCDIC encoding considerations

There are a couple of EBCDIC encoding considerations to deal with when trying to parse an XML file on z/OS. The first involves the character set differences between EBCDIC and Unicode. Because only a small number of Unicode characters can be represented in EBCDIC, when an EBCDIC encoded XML document is parsed, any Unicode character entity in the parsed document that does not have an EBCDIC value is converted into a dash.

Note: The default for an non-representable character is a dash. This can be overridden with a control call to XEC_CTL_ENTS_AND_REFS.

Secondly, if the EBCDIC XML document has been created or modified on a z/OS system, then the line ending character is typically a NL (x'15') character. This is commonly associated with the Unicode NEL character (x'85'). For EBCDIC code page documents, the z/OS XML parser will accept XML 1.0 documents that have a NL as a line termination character, and will normalize all line-endings to EBCDIC NL (NEL). However, because these documents are non-compliant, they may not be accepted by parsers on other platforms. In general, EBCDIC is not a portable encoding so IBM does not recommend using EBCDIC for XML documents going between platforms or on the Internet.

Note: For XML 1.1 documents, NL is legitimate and the z/OS XML parser is compliant in processing it as such.

Managing memory resources

The z/OS XML parser processes a document using memory resources that are provided by the caller. This storage is passed from caller to z/OS XML parser in the form of a Parse Instance Memory Area (PIMA). This required data area is used by the z/OS XML parser to suballocate a call stack, control blocks, and the tables and trees that are used to hold assorted document-specific information for the document being parsed. The environment created by the z/OS XML parser in this memory area completely describes the context of a given document parse.

A memory allocation exit is supported by the z/OS XML parser so that the caller can provide a pair of allocation/deallocation services. The allocation service will be called by the z/OS XML parser in the event that a given document causes the z/OS XML parser to exhaust the PIMA. For performance reasons, it is best if the PIMA provided by the invoker is large enough that this exit is not used. However, the exit gives the z/OS XML parser a means to complete processing of a document in the event that the memory area provided at initialization time is too small. This exit is only used to extend the PIMA, and is not used in any way to manage input or output buffers.

The deallocation service will be called by the z/OS XML parser to free the memory extension created by the allocation service. The deallocation service will never free the original PIMA storage.

For callers that do not provide a memory allocation exit, the z/OS XML parser provides default routines to allocate and free memory. The z/OS XML parser also provides an option at initialization time allowing the caller to specify how the z/OS XML parser's default routine allocates memory. This feature should be specified when PIMAs are used on multiple tasks, in order to prevent task termination from causing storage extents to be freed before the z/OS XML parser is done using them. Normally, z/OS XML parser will allocate memory at the task level. However, when the feature is specified, the z/OS XML parser will allocate memory at the Job Step Task (JST) level instead.

In both cases, the caller is assuming the responsibility to call GXL1TRM (GXL4TRM) in the event the z/OS XML parser abends and the caller's recovery gets control.

When no memory allocation exit is provided, the subpool used will be as follows:

- If running in SRB or cross memory mode, subpool 129 will be used. This is JST related and cannot be freed by unauthorized callers. The key will be the same as the key at the time the z/OS XML parser is invoked.
- If running in task mode (PSATOLD not zero), with PRIMARY=SECONDARY=HOME, then the subpool chosen will depend on the authorization state of the caller and on the specification of the XEC_FEAT_JST_OWNS_STORAGE feature on the GXL1INI (GXL4INI) call. If the caller is running in key 0-7 or supervisor state, they will be considered authorized.
  - Authorized and JST requested — subpool 129
  - Authorized and JST not requested — subpool 229
  - Unauthorized and JST requested — subpool 131
  - Unauthorized and JST not requested — subpool 0

    Note: If running on a subtask which is sharing subpool 0, then this storage will be owned by the task that owns subpool 0.

These choices of subpool will eliminate the possibility of the z/OS XML parser running in an authorized state while using problem key storage which could be freed and reallocated.

Using return and reason codes

The z/OS XML parser API services provide a return and reason code to indicate the success or failure of the parse process. The return code is a fullword value that indicates the class of the return status, and takes on one of the following values:

- Success (XRC_SUCCESS)
- Warning (XRC_WARNING) - parsing is successful, but incomplete. This is most often caused by the z/OS XML parser reaching the end of either the input or the output buffer.
- Failure (XRC_FAILURE) - a terminating failure has occurred. The return information passed back in the parameters, such as the numbers of bytes left in the input and output buffers, are valid. The extended diagnostic information may also contain additional problem determination information that is of use.
- Not-well-formed (XRC_NOT_WELL_FORMED) - a terminating failure has occurred because the input document is not well formed. As with the failure case above, all return information passed back through the parameters and extended diagnostic area is valid.
- Fatal (XRC_FATAL) - a terminating error has occurred. None of the return information is valid.

- Not valid (XRC_NOT_VALID) - The document is not valid according to the specified schema.

In addition to the return code describing the class of error, the reason code provides more detail. The reason code is only valid when the return code is not XRC_SUCCESS. When a service of the z/OS XML parser API returns XRC_SUCCESS, the reason code may have any random value.

The reason code itself is a fullword value, but is made up of two halfwords. The upper halfword is reserved for a module identifier that is used by IBM Service to isolate the source of the problem, and the lower halfword indicates the reason why the parse process was paused or terminated. When checking the value of the reason code, the caller must be sure to AND the reason code with the reason code mask (XRSN_REASON_MASK) before testing the value. The declaration of XRSN_REASON_MASK and all of the defined reason code values are contained in the GXLYXR macro. A list of the reason codes and their descriptions can be found in Appendix B, “Reason codes listed by value,” on page 161.

Chapter 6. z/OS XML parser API: C/C++

This chapter lists the C/C++ callable services interface used for the z/OS XML parser.

Setting the XPLINK(ON) Language Environment runtime option

If the calling application is compiled without XPLINK and wants to use z/OS XML System Services, the calling application must set the following option:

```
export _CEE_RUNOPTS="XPLINK(ON)"
```

If this option is not set, an error will occur once the application is run.

For more information on the XPLINK compiler option, see z/OS Language Environment Programming Guide.

Support for the Metal C compiler option

Support is provided for callers who wish to use the Metal C compiler option. The same APIs available to the standard C and C++ callers are also available to Metal C users, with the following restrictions:

- All parameters must be variables.
- The functions do not return values.

  Note: Return codes and reason codes are still returned through the parameter lists.

For more information on how to use the Metal C compiler option, see *Metal C Run-time Library Guide and Reference*.

Where to find the header files, DLLs and side decks

Header files for non-Metal C can be found in the z/OS UNIX directory `/usr/include`. Header files for Metal C can be found in the z/OS UNIX directory `/usr/include/metal`. If you are not using z/OS UNIX, then the non-Metal C header files can be found in the PDSE SYS1.SIEAHDRV.H . There are no Metal C header files for the batch environment.

DLLs for non-Metal C can be found in the z/OS UNIX directory `/usr/lib`. If you are not using z/OS UNIX, then the DLLs can be found in SYS1.SIEALNKE . There are no DLLs for Metal C.

Side decks for non-Metal C can be found in `/usr/lib`. If you are not using z/OS UNIX, then the side decks can be found in SYS1.SIEASID. There are no side decks for Metal C.

Using the recovery routine

z/OS XML provides an ARR recovery routine to assist with problem determination and diagnostics. In the C/C++ environment, the recovery routine is provided as the default setting in most cases and will recover the code and collect dumps for most abends that occur during a parse. For unauthorized C/C++ callers, an

IEATDUMP will be taken in data set *userid*.GXLSCXML.DYYMMDD.THHMMSS.DUMP, where the *userid* is extracted from the task level ACEE if present or the address space ACEE, and where DYYMMDD is the date and THHMMSS is the time the dump was taken. For authorized C/C++ callers, an SDUMPX will be taken into a system dump data set. See “ARR recovery routine” on page 158 for more information.

In order to effectively use the recovery routine, you must set the following runtime option: TRAP(ON,NOSPIE). If this runtime option is not set, unpredictable behavior may result with regard to recovery.

z/OS XML XL C/C++ API

gxlpControl — perform a parser control function

Description

This is a general purpose service which provides control functions for interacting with the z/OS XML parser. The function performed is selected by setting the *ctl_option* parameter using the constants defined in gxlhxec.h . These functions include:

GXLHXEC_CTL_FIN
The caller has finished parsing the document. Reset the necessary structures so that the PIMA can be reused on a subsequent parse, and return any useful information about the current parse. For more information on this function, see “GXLHXEC_CTL_FIN” on page 57.

GXLHXEC_CTL_FEAT
The caller wants to change the feature flags. A GXLHXEC_CTL_FIN function will be done implicitly.

Note: Some feature flags are not supported on gxlpControl. See “GXLHXEC_CTL_FEAT” on page 58 for a list of these feature flags.
For more information on this function, see “GXLHXEC_CTL_FEAT” on page 58.

GXLHXEC_CTL_LOAD_OSR
The caller wants to load and use an Optimized Schema Representation (OSR) for a validating parse. For more information on this function, see “GXLHXEC_CTL_LOAD_OSR” on page 60.

GXLHXEC_ CTL_QUERY_MIN_OUTBUF
The caller is requesting the minimum output buffer size required on a subsequent parse. This function will also enable the parse to be continued after a GXLHXRSN_BUFFER_OUTBUF_SMALL reason code has been received from gxlpParse.

Note: Finish and reset processing is performed by all operations available through this control service, except GXLHXEC_CTL_QUERY_MIN_OUTBUF and GXLHXEC_CTL_LOAD_OSR. See the descriptions of these operations under ctl_operation for more information.
For more information on this function, see “GXLHXEC_CTL_QUERY_MIN_OUTBUF” on page 62.

GXLHXEC_CTL_ENTS_AND_REFS

The caller can request additional flexibility when processing character and entity references as follows:

- When an unresolved entity reference is encountered, the caller can request that the parser stop processing and return an error record.
- When a character reference which cannot be represented in the current code page is encountered, z/OS XML System Services places a dash (-) in the output stream for that character. The caller may specify, with this control call, to output a character other than dash (-) in the output stream.
- When a character reference which cannot be represented in the current code page is encountered, the caller can request, using this control call, an additional output record to be generated in the output stream that contains information about this character reference.

Note:

1. Finish and reset processing is performed for this control operation. See "Usage notes" on page 116for more information.
2. If the parse instance has been initialized to process XDBX binary XML streams, then the input stream will never have entity references to resolve. Performing the GXLHXEC_CTL_ENTS_AND_REFS operation will have no effect on the output of the parser. In order to prevent accidental attempted use of this operation in this environment, the parser will return a failure for this control request if the input is an XDBX stream.

For more information on this function, see "GXLHXEC_CTL_ENTS_AND_REFS" on page 63.

GXLHXEC_CTL_LOAD_FRAG_CONTEXT

The caller wants to load fragment context including fragment path and namespace binding information for document fragment parsing.

Note:

1. This control operation does not perform finish and reset processing through the control service. See the description in ctl_operation for more information.
2. Fragment parsing is not supported for XDBX input. For this reason, attempting to load a fragment context for parse instances initialized to handle XDBX streams will fail.

For more information on this function, see "GXLHXEC_CTL_LOAD_FRAG_CONTEXT" on page 64.

GXLHXEC_CTL_FRAGMENT_PARSE

The caller wants to enable or disable document fragment parsing.

Note:

1. This control operation does not perform finish and reset processing through the control service. See the description in ctl_operation for more information.
2. Fragment parsing is not supported for XDBX input. For this reason, attempting to enable document fragment parsing for parse instances initialized to handle XDBX streams will fail.

For more information on this function, see "GXLHXEC_CTL_FRAGMENT_PARSE" on page 66.

GXLHXEC_CTL_RESTRICT_ROOT
The caller wishes to restrict the root element name on the next parse. This operation is only valid when the PIMA has been configured for validation and schema information is requested. For more information on this function, see "GXLHXEC_CTL_RESTRICT_ROOT" on page 69.

GXLHXEC_CTL_ERROR_HANDLING
With this control operation, the caller can do the following for a validating parse:

- Enable the creation of auxiliary records which can include the location of an error in the XML document, the string which is in error, and also a possible expected string.
- Enable position indexes to be present in the error location path in order to facilitate locating the error.

For a non-validating parse, it can be used to:

- Enable the ability to continue parsing when an undefined prefix is encountered on an element or attribute. The "prefix:local name" will be treated as the local name.
- Request an auxiliary information record that contains the tolerated return and reason codes and the error offset.

For more information on this function, see "GXLHXEC_CTL_ERROR_HANDLING" on page 71.

Performance Implications

The finish-and-reset function allows the caller to re-initialize the PIMA to make it ready to handle a new XML document. This re-initialization path enables the z/OS XML parser to preserve its existing symbol table, and avoid other initialization pathlength that's performed by calling the initialization service. The reset features function also allows the caller to re-initialize the z/OS XML parser as above and allows the feature flags to be reset as well.

Usage notes

This callable service is mapped to GXL1CTL (GXL4CTL). Refer to"Usage notes" on page 116 of GXL1CTL (GXL4CTL) for usage information. For a list of properties and resources reset by the control functions, see "Properties and resources reset by control functions."

Properties and resources reset by control functions

When the control functions are utilized by a caller (the GXL1CTL (GXL4CTL) API is invoked), some of the z/OS XML parser properties and resources are reset while others are not. The properties and resources reset and by which control functions are shown in the following table. Properties and resources not reset by a particular control function may need to be explicitly restored by a PAB copy.

Table 24. z/OS XML parser properties and resources reset by control functions

Properties and resources	Control functions that reset
Loaded OSRs, XML fragment contexts, and allowable root names	None

Table 24. z/OS XML parser properties and resources reset by control functions (continued)

Properties and resources	Control functions that reset
The following control and initialization settings (listed by features) XEC_FEAT_STRIP_COMMENTS, XEC_FEAT_TOKENIZE_WHITESPACE, XEC_FEAT_CDATA_AS_CHARDATA, XEC_FEAT_SOURCE_OFFSETS, XEC_FEAT_FULL_END	XEC_CTL_FIN
Entity resources	XEC_CTL_FIN
The following system level resources: recovery status, JST owns storage, z/OS XML System Services exit routines	None
Parser types (validating and nonvalidating)	None
Fragment mode	XEC_CTL_FIN. XEC_CTL_FEAT, XEC_CTL_LOAD_OSR, XEC_CTL_ENTS_AND_REFS, XEC_CTL_ERROR_HANDLING
Start of the XML document	XEC_CTL_FIN. XEC_CTL_FEAT, XEC_CTL_LOAD_OSR, XEC_CTL_ENTS_AND_REFS, XEC_CTL_ERROR_HANDLING
Error state	XEC_CTL_FIN. XEC_CTL_FEAT, XEC_CTL_LOAD_OSR, XEC_CTL_ENTS_AND_REFS, XEC_CTL_ERROR_HANDLING, XEC_CTL_QUERY_MIN_OUTBUF (only when RC=8, RSN= XRSN_BUFFER_OUTBUF_SMALL)

gxlpControl features and functions

GXLHXEC_CTL_FIN

Description

This indicates that the caller wishes to end the current parse at the current position in the XML document. The PIMA is re-initialized to allow it to be used on a new parse request. To free up all resources associated with the parse instance, the caller should use the termination service. If the caller issues this control operation after document fragment parsing is enabled, then this control operation will disable document fragment parsing and re-initialize the PIMA for a new parse request. The loaded fragment context will remain in storage and become active when fragment mode is enabled.

Syntax

```
int gxlpControl (void * PIMA,
                int ctl_operation,
                void * ctl_data_p,
                int * rc_p,
                int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_operation

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_FIN.

ctl_data_p

Supplied and returned parameter

Type: void *

The name of the parameter that contains the address where the service will store the address of the diagnostic area, which is mapped by header file gxlhxd.h . This provides additional information that can be used to debug problems in data passed to the z/OS XML parser. The diagnostic area resides within the PIMA, and will be overlaid on the next call to the z/OS XML parser. If the caller does not wish receive diagnostic information, the NULL value is used in place of the address of the diagnostic area.

rc_p

Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Example

```
void *PIMA;
GXLHXD *ctlDiagArea = NULL;
void *CTL_dataArea = ctlDiagArea;
int lastRetVal, lastRC, lastRSN;
lastRetVal = gxlpControl(PIMA,
                        GXLHXEC_CTL_FIN,
                        &CTL_dataArea,
                        &lastRC,
                        &lastRSN);
```

GXLHXEC_CTL_FEAT

Description

This indicates that the caller wishes to re-initialize the z/OS XML parser, as with the reset-and-finish function above, and in addition, that the caller wishes to reset some of the feature flags used during the parse.

Note: The following feature flags are not supported by this service:

- GXLHXEC_FEAT_JST_OWNS_STORAGE
- GXLHXEC_FEAT_RECOVERY
- GXLHXEC_FEAT_VALIDATE
- GXLHXEC_FEAT_SCHEMA_DISCOVERY
- GXLHXEC_FEAT_XDBX_INPUT

Make sure that these feature flags are turned to the OFF state before calling this service to set the feature flags. If these features need to be changed (for example, if switching between validating and non-validating parses), the parse instance must be terminated and re-initialized with the required feature settings.

Syntax

```
int gxlpControl (void * PIMA,
                 int ctl_operation,
                 void * ctl_data_p,
                 int * rc_p,
                 int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_operation

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_FEAT.

ctl_data_p

Supplied and returned parameter

Type: void *

This parameter must contain the address of a fullword (doubleword), which is mapped by header file gxlhxft.h. See "gxlhxft.h (GXLYXFT) - mapping of the control feature input output area" on page 218 for more information on this header file.

The GXLHXFT_FEAT_FLAGS parameter is an input parameter to the API and contains the value of feature flags to be used in the subsequent parse. It is defined as follows:

GXLHXEC_FEAT_STRIP_COMMENTS

This effectively strips comments from the document by not returning any comments in the parsed data stream. Default: off.

GXLHXEC_FEAT_TOKENIZE_WHITESPACE

This sets the default token value for white space preceding markup in the root element to an explicit white space value. Default: off – white space is returned as character data.

GXLHXEC_FEAT_CDATA_AS_CHARDATA
This returns CDATA in records with a CHARDATA token type. The content of these records may contain text that would normally have to be escaped to avoid being handled as markup. Default: off.

GXLHXEC_FEAT_SOURCE_OFFSETS
This feature is used to include records in the parsed data stream which contain offsets to the corresponding structures in the input document. Default: off.

GXLHXEC_FEAT_FULL_END
This feature is used to expand the end tags to include the local name, prefix and URI corresponding to the qname on the end tag. Default: off.

If none of the features are required, pass the name of a fullword field containing zero. Do not construct a parameter list with a zero pointer in it.

rc_p
Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p
Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Example

```
void *PIMA;
int lastRetVal, lastRC, lastRSN;
GXLHXFT ft;
ft.XFT_FEAT_FLAGS=0;
void *pft = &ft;
lastRetVal = gxlpControl(PIMA,
                         GXLHXEC_CTL_FEAT,
                         &pft,
                         &lastRC,
                         &lastRSN);
```

GXLHXEC_CTL_LOAD_OSR

Description

This indicates that the caller wants to load and use a given Optimized Schema Representation (OSR) during a validating parse. If the parse prior to invoking this operation returned a GXLHXRSN_NEED_OSR, this operation will not perform reset and finish processing.

Syntax

```
int gxlpControl (void * PIMA,
                 int ctl_operation,
                 void * ctl_data_p,
                 int * rc_p,
                 int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_operation

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_LOAD_OSR.

ctl_data_p

Supplied and returned parameter

Type: void *

This indicates that the caller wants to load and use a given Optimized Schema Representation (OSR) during a validating parse. Once an OSR has been loaded, it remains in use for all validating parse requests until a different OSR is provided by calling this service again.

This parameter must contain the address of an area containing information about the OSR to load. This area is mapped by gxlhxosr.h. See “gxlhxosr.h (GXLYXOSR) - mapping of the OSR control area” on page 218 for more information on the structures in this header.

rc_p

Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Example

```
void *PIMA;
GXLHXOSR *ctlData;
int lastRetVal, lastRC, lastRSN;
lastRetVal = gxlpControl(PIMA,
                         GXLHXEC_CTL_LOAD_OSR,
                         (void *)&ctlData,
                         &lastRC,
                         &lastRSN);
```

GXLHXEC_CTL_QUERY_MIN_OUTBUF

Description

This indicates that the caller is requesting the control service to return the minimum output buffer size required for subsequent parse to complete without returning an GXLHXRSN_BUFFER_OUTBUF_SMALL reason code. This value is returned in the XD control block.

Syntax

```
int gxlpControl (void * PIMA,
                 int ctl_operation,
                 void * ctl_data_p,
                 int * rc_p,
                 int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_operation

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_QUERY_MIN_OUTBUF.

ctl_data_p

Supplied and returned parameter

Type: void *

This parameter must contain the address of a fullword (doubleword) where the service will store the address of the diagnostic area, which is mapped by header file gxlhxd.h. The field XD_MIN_OB contains the minimum output buffer size required on the next parse. If some failure other than GXLHXRSN_BUFFER_OUTBUF_SMALL occurred prior to this call, GXLHXRSN_CTL_SEQUENCE_INCORRECT will be returned. The XD area will not be returned.

The diagnostic area resides within the PIMA, and will be overlaid on the next call to the z/OS XML parser.

rc_p

Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

GXLHXEC_CTL_ENTS_AND_REFS

Description

This indicates that the caller is requesting additional flexibility when processing character or entity references. When this option is specified, the *ctl_data_p* parameter must also be utilized to specify the specific enhancement being requested. See the *ctl_data_p* section below for more information.

Syntax

```
int gxlpControl (void * PIMA,
                 int ctl_operation,
                 void * ctl_data_p,
                 int * rc_p,
                 int * rsn_p);
```

Parameters

`PIMA`

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

`ctl_operation`

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_ENTS_AND_REFS.

`ctl_data_p`

Supplied and returned parameter

Type: void *

This parameter must contain the address of an area that contains information about what reference operations are to be processed. This area is mapped by the XEAR data structure in file gxlhctl.h.

`rc_p`

Returned parameter

Type: int *

The name of the area where the service stores the return code.

`rsn_p`

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Example

```
void *PIMA;
GXLHXEAR ear;
ear.XEAR_VERSION=1;
void *CTL_ear_p = &ear;
int lastRetVal, lastRC, lastRSN;
lastRetVal = gxlpControl(PIMA,
                         GXLHXEC_CTL_ENTS_AND_REFS,
                         &CTL_ear_p,
                         &lastRC,
                         &lastRSN);
```

GXLHXEC_CTL_LOAD_FRAG_CONTEXT

Description

This indicates that the caller wants to load fragment context into the z/OS XML parser. This service allows the caller to load namespace binding information and fragment paths for document fragment parsing. Namespace binding information is optional. Fragment path is required . This service must be issued prior to a GXLHXEC_CTL_FRAGMENT_PARSE control operation that enables document fragment parsing. If fragment context is already loaded from a prior GXLHXEC_CTL_LOAD_FRAG_CONTEXT control operation and this service is called again, the new fragment context will overlay the previously loaded context. This control operation will not cause finish/reset processing to take place.

Syntax

```
int gxlpControl (void * PIMA,
                 int ctl_operation,
                 void * ctl_data_p,
                 int * rc_p,
                 int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_operation

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_LOAD_FRAG_CONTEXT.

ctl_data_p

Supplied and returned parameter

Type: void *

This parameter must contain a pointer to where the service will locate the address of the document fragment context structure, which is mapped by the header gxlhctl.h. The name of the data structure is GXLHXFC. This structure allows the caller to provide the fragment path and namespace binding information to assist document fragment parsing.

To validate an element during document fragment parsing, the fragment path represents the path from the root element of the complete document to the root element of the fragment, which consists of prefixes and localnames. To validate an attribute during fragment parsing, the fragment path represents the path from the root element of the complete document to the desired attribute name. The fragment path is required in order to perform validation in fragment parsing.

The fragment path syntax is defined below:

```
FragmentPath ::= ('/' ElementName)* FragmentData
FragmentData ::= '/' ElementName ('/@' AttributeName)?
ElementName ::= QName
AttributeName ::= QName
```

Namespaces bindings allow unique strings of text that identify a given space of names to be represented by a prefix. This allows references to elements with the same name to be differentiated, based on the namespace to which they belong. These bindings may not be present in the document fragment, and often these bindings exist in the ancestor elements' start tag that is not part of the document fragment. The caller can provide a complete context containing multiple namespace bindings in the GXLHXFC structure. The namespace binding is optional information.

However, if there is an XML instance document that uses a default namespace, the caller must still specify a prefix on the element names in the fragment path. The caller must also specify this prefix along with the namespace URI in the namespace binding information. The actual prefix does not matter; only the namespace URI matters, but the prefix will associate each element in the fragment path with the correct namespace.

Note:

1. All the strings for fragment path and namespace binding passed into the GXLHXEC_CTL_LOAD_FRAG_CONTEXT control call needs to be in the encoding of the z/OS XML parser configured at initialization time.
2. If the caller disables document fragment parsing, the namespace contexts loaded through the GXLHXEC_CTL_LOAD_FRAG_CONTEXT control call will be removed and will not be available during the non-fragment parsing mode.
3. When the caller issues a GXLHXEC_CTL_LOAD_FRAG_CONTEXT control call to load namespace contexts, the namespace contexts will be available when the z/OS XML parser switches into fragment parsing mode. The namespace contexts will only get unloaded and replaced if the caller terminates the parser or issues GXLHXEC_CTL_LOAD_FRAG_CONTEXT control call again to load new namespace contexts.

rc_p

Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Example

```
void * PIMA;
void * fragContext;
void * fragParse;
void * ctl_data_p;
int * option_flags;
void * fragbuf; int fragbuf_left;
void * outbuf; int outbuf_left;
GXLHXFP xfp;
GXLHXFC xfc;
GXLHXFC_ENTRY xfc_entry[1];
char * nspfx_str; char * nsuri_str;
char * fragPath;
int rc, rsn;
/* Perform necessary setup */
nspfx_str = "ibm";
nsuri_str = "http://w3.ibm.com";
fragPath = "/ibm:root/ibm:person";
/* Perform a reset */
gxlpControl(PIMA,
            GXLHXEC_CTL_FIN,
            &ctl_data_p,
            rc,
            rsn);
/* setup the GXLHXFC structure with namespace binding information */
memset(&xfc,0,sizeof(GXLHXFC));
xfc.XFC_ENTRY_NSCOUNT = 1;
xfc_entry[0].XFC_ENTRY_NSPFX_LEN = strlen(nspfx_str);
xfc_entry[0].XFC_ENTRY_NSPFX_PTR = nspfx_str;
xfc_entry[0].XFC_ENTRY_NSURI_LEN = strlen(nsuri_str);
xfc_entry[0].XFC_ENTRY_NSURI_PTR = nsuri_str;
xfc.XFC_ENTRY_NS_PTR = &xfc_entry
xfc.XFC_FRAGPATH_PTR = fragPath;
xfc.XFC_FRAGPATH_LEN = strlen(fragPath);
fragContext = (void*)&xfc
/* initialize the GXLHXFP structure with zero and set the enable flag */
memset(&xfp,0,sizeof(GXLHXFP));
xfp.XFP_FLAGS = XFP_FLAGS_FRAGMENT_MODE;
fragParse = (void*)&xfp
/* Load the fragment parsing contexts */
gxlpControl(PIMA,
            GXLHXEC_CTL_LOAD_FRAG_CONTEXT,
            &fragContext,
            rc,
            rsn);
```

GXLHXEC_CTL_FRAGMENT_PARSE

Description

This indicates that the caller wants to either enable or disable document fragment parsing. This service will decide whether to enable or disable document fragment parsing based on the XFP_FLAGS_FRAGMENT_MODE bit set in the *ctl_data_p* parameter. Document fragment parsing is disabled by default. This control operation will not cause finish/reset processing to take place. If the caller wants to parse a new complete XML document, a GXLHXEC_CTL_FIN control operation must be called prior to a new parse request. If any error with return code greater than 4 has occurred during document fragment parsing, a GXLHXEC_CTL_FIN control operation must be issued in order to resume parsing. Calling the GXLHXEC_CTL_FIN control operation will disable the document fragment parsing and unload all fragment contexts.

Note:

1. Document fragment parsing can only be enabled once before disabling. Likewise, document fragment parsing can only be disabled once before enabling.
2. If the caller disables document fragment parsing, the parse will end and the caller is allowed to parse a new document.

Syntax

```
int gxlpControl (void * PIMA,
                 int ctl_operation,
                 void * ctl_data_p,
                 int * rc_p,
                 int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_operation

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_FRAGMENT_PARSE.

ctl_data_p

Supplied and returned parameter

Type: void *

This parameter must contain a pointer to where the service will locate the address of the document fragment parsing structure, which is mapped by the header gxlhctl.h. The name of the data structure is GXLHXFP. This structure allows the caller to specify whether to enable or disable document fragment parsing through the XFP_FLAGS_FRAGMENT_MODE bit set in the XFP_FLAGS field. Document fragment parsing is disabled by default.

The XFP_XD_PTR is where the service will store the address of the diagnostic area, which is mapped by macro GXLYXD. This provides additional information that can be used to debug problems in data passed to the z/OS XML parser. The diagnostic area resides within the PIMA, and will be overlaid on the next call to the z/OS XML parser.

Tips:

- To enable document fragment parsing, set the XFP_FLAGS_FRAGMENT_MODE bit to on.
- To disable document fragment parsing, set the XFP_FLAGS_FRAGMENT_MODE bit to off.

Note:

1. When the caller validates an attribute during fragment parsing, the document fragment passed to the parser should contain only the desired attribute's value.
2. When the caller re-enables document fragment parsing after it has been disabled, and without calling load fragment context again, the previous loaded fragment context will be utilized in this new fragment parse. This includes the fragment path and any namespace binding information.
3. The OSR must be loaded by way of the XEC_CTL_LOAD_OSR control call prior to enabling fragment parsing.

rc_p
Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p
Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Example

```
void * PIMA
void * fragContext;
void * fragParse;
void * ctl_data_p;
int * option_flags;
void * fragbuf; int fragbuf_left;
void * outbuf; int outbuf_left;
GXLHXFP xfp;
GXLHXFC xfc;
GXLHXFC_ENTRY xfc_entry[1];
char * nspfx_str; char * nsuri_str;
char * fragPath;
int rc, rsn;
/* Perform necessary setup */
nspfx_str = "ibm";
nsuri_str = "http://w3.ibm.com";
fragPath = "/ibm:root/ibm:person";
/* Perform a reset */
gxlpControl(PIMA,
            GXLHXEC_CTL_FIN,
            &ctl_data_p,
            rc,
            rsn);
/* setup the GXLHXFC structure with namespace binding information */
memset(&xfc,0,sizeof(GXLHXFC));
xfc.XFC_ENTRY_NSCOUNT = 1;
xfc_entry[0].XFC_ENTRY_NSPFX_LEN = strlen(nspfx_str);
xfc_entry[0].XFC_ENTRY_NSPFX_PTR = nspfx_str;
xfc_entry[0].XFC_ENTRY_NSURI_LEN = strlen(nsuri_str);
xfc_entry[0].XFC_ENTRY_NSURI_PTR = nsuri_str;
xfc.XFC_ENTRY_NS_PTR = &xfc_entry
xfc.XFC_FRAGPATH_PTR = fragPath;
xfc.XFC_FRAGPATH_LEN = strlen(fragPath);
fragContext = (void*)&xfc
/* initialize the GXLHXFP structure with zero and set the enable flag */
memset(&xfp,0,sizeof(GXLHXFP));
```

```
xfp.XFP_FLAGS = XFP_FLAGS_FRAGMENT_MODE;
fragParse = (void*)&xfp
/* Load the fragment parsing contexts */
gxlpControl(PIMA,
            GXLHXEC_CTL_LOAD_FRAG_CONTEXT,
            &fragContext,
            rc,
            rsn);
/* Note: the OSR must be loaded at this point */
/* Enable document fragment parsing */
gxlpControl(PIMA,
            GXLHXEC_CTL_FRAGMENT_PARSE,
            &fragParse,
            &rc,
            &rsn);
/* Parse the desired document fragments */
gxlpParse(PIMA,
          option_flags,
          &fragbuf,
          &fragbuf_left,
          &outbuf,
          &outbuf_left,
          &rc,
          &rsn);
/* Disable document fragment parsing */
xfp.XFP_FLAGS = 0;
gxlpControl(PIMA,
            GXLHXEC_CTL_FRAGMENT_PARSE,
            &fragParse,
            &rc,
            &rsn);
```

GXLHXEC_CTL_RESTRICT_ROOT

Description

This operation indicates that the caller wishes to restrict the root element name on the next parse. If the root element name is not any of those listed in the GXLHXRR data area, this call will cause the parse to stop. This operation will reset the PIMA.

Syntax

```
int gxlpControl (void * PIMA,
                 int ctl_operation,
                 void * ctl_data_p,
                 int * rc_p,
                 int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_operation

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_RESTRICT_ROOT.

ctl_data_p
Supplied and returned parameter

Type: void *

This parameter contains the address of an area with information about the restricted root element. This area is mapped by the header file gxlhxrr.h. This provides a list of names that must contain the name of the root element in order for the validating parse to succeed.

rc_p
Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p
Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Example

```
void * PIMA;
void * ctl_data_p;
void * rootTable;
void * inbuf; int inbuf_left;
void * outbuf; int outbuf_left;
GXLHXRR_ENTRY *entry;
int * option_flags;
GXLHXRR xrr;
char * root_str; char * rootnsuri_str;
int rc, rsn;
/* Perform necessary setup */
root_str = "personal";
rootnsuri_str = "http://w3.ibm.com";
/* Perform a reset */
gxlpControl(PIMA,
            GXLHXEC_CTL_FIN,
            &ctl_data_p,
            rc,
            rsn);
/* setup the GXLHXRR structure with namespace binding information */
memset(&xrr,0,sizeof(GXLHXRR));
xrr.XRR_ENTRY_COUNT = 1;
entry.XRR_ENTRY_ROOT_LEN = strlen(root_str);
entry.XRR_ENTRY_ROOT_PTR = root_str;
entry.XRR_ENTRY_NSURI_LEN = strlen(rootnsuri_str);
entry.XRR_ENTRY_NSURI_PTR = rootnsuri_str;
xrr.XRR_ENTRY = entry;
rootTable = (void*)&xrr
/* Enable Root Restriction */
gxlpControl(PIMA,
            GXLHXEC_CTL_RESTRICT_ROOT,
            &rootTable,
            rc,
            rsn);
/* Parse the desired document fragments */
gxlpParse(PIMA,
          option_flags,
          &inbuf,
         &inbuf_left,
         &outbuf,
         &outbuf_left,
         &rc,
         &rsn);
/* Disable document fragment parsing */
xrr.XRR_ENTRY_COUNT = 0;
gxlpControl(PIMA,
            GXLHXEC_CTL_RESTRICT_ROOT,
            &rootTable,
            &rc,
            &rsn);
```

GXLHXEC_CTL_ERROR_HANDLING

Description

With this control operation, the caller can do the following for a validating parse:

- Enable the creation of auxiliary records which can include the location of an error in the XML document, the string which is in error, and also a possible expected string.
- Enable position indexes to be present in the error location path in order to facilitate locating the error.

For a non-validating parse, it can be used to:

- Enable the ability to continue parsing when an undefined prefix is encountered on an element or attribute. The “prefix:local name” will be treated as the local name.
- Request an auxiliary information record that contains the tolerated return and reason codes and the error offset.

Syntax

```
int gxlpControl (void * PIMA,
                 int ctl_operation,
                 void * ctl_data_p,
                 int * rc_p,
                 int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_operation

Supplied parameter

Type: int

The name of the parameter containing an integer value initialized to GXLHXEC_CTL_ERROR_HANDLING.

ctl_data_p

Supplied and returned parameter

Type: void *

This parameter contains the address of an area with information about the error string. This is the XERR data structure which is mapped by GXLHERR in the header file gxlhctl.h.

rc_p

Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

The enhanced error information for a validating parse is returned by way of the XERR_XD_PTR and is where the service will store the address of the diagnostic area, which is in gxlhxd.h file. The XD_LastOutput field is a pointer to the data area containing these records. This data area is within the PIMA and is formatted in the same manner as a normal output buffer.

The XEC_TOLERATED_ERROR auxiliary info record for a non-validating parse is returned in the output buffer. In the event that source offset auxiliary records are also being returned, this record will immediately follow those records for the element or attribute in the output buffer.

In addition to enabling or disabling the enhanced error features, this control option will perform a reset function. The following properties and resources will be reset by this control option:

- Fragment mode (validating parse only)
- Start of the XML document
- Error state

gxlpInit — initialize the z/OS XML parser

Description

The gxlpInit callable service initializes the PIMA and records the addresses of the caller's system service routines (if any). The PIMA storage is divided into the areas that will be used by the z/OS XML parser to process the input buffer and produce the parsed data stream.

Performance Implications

The initialization of structures used by the z/OS XML parser in the PIMA is only done once per parse and is therefore unlikely to affect performance. The caller may choose to reuse the PIMA after each parse to eliminate the overhead of storage allocation and the page faults that occur when referencing new storage. In this case, a control operation is required to reset the necessary fields in the PIMA before parsing can continue. For more information on the control operation, see “gxlpControl — perform a parser control function” on page 54.

Syntax

```
int gxlpInit (void * PIMA,
              long PIMA_LEN,
              int ccsid,
              int feature_flags,
              GXLHXSV sys_svc_vector,
              void * sys_svc_parm,
              int * rc_p,
              int * rsn_p);
```

Parameters

PIMA

Pointer to Parse Instance Memory Area (PIMA).

Type: void *

PIMA_Len

Length of PIMA

Type: long

The name of an area containing the length of the Parse Instance Memory Area. This service validates the length of this area against a minimum length value. The minimum length of the PIMA depends on whether or not validation will be performed during the parse:

- GXLHXEC_NVPARSE_MIN_PIMA_SIZE (non-validating)
- GXLHXEC_VPARSE_MIN_PIMA_SIZE (validating)

ccsid

Supplied parameter

Type: Integer

The Coded Character Set IDentifier (CCSID) that identifies the document's character set. The CCSID value in this parameter will override any character set or encoding information contained in the XML declaration of the document. A set of CCSID constants for supported encodings has been declared in GXLYXEC. See Appendix I, "Supported encodings," on page 239 for a full list of supported encodings.

feature_flags

Supplied parameter

Type: Integer

The name of the area that contains an integer value representing one or more of the following z/OS XML parser features. OR these flags together as needed to enable features. Choose any of the following:

- **GXLHXEC_FEAT_CDATA_AS_CHARDATA** - return CDATA in records with a CHARDATA token type. The content of these records may contain text that would normally have to be escaped to avoid being handled as markup.
- **GXLHXEC_FEAT_FULL_END** - expand the end tags to include the local name, prefix and URI corresponding to the qname on the end tag.
- **GXLHXEC_FEAT_JST_OWNS_STORAGE** - allocate storage as Job Step Task (JST) related instead of task related. See the "Usage notes" on page 134 below for more information.

- **GXLHXEC_FEAT_RECOVERY** - this option is used to turn on the recovery routine.

  Note: Because the recovery routine is automatically enabled for Language Environment-C, this option is only meaningful when using the Metal C compiler option.
- **GXLHXEC_FEAT_SOURCE_OFFSETS** - include records in the parsed data stream which contain offsets to the corresponding structures in the input document.
- **GXLHXEC_FEAT_STRIP_COMMENTS** - effectively strip comments from the document by not returning any comments in the parsed data stream.
- **GXLHXEC_FEAT_TOKENIZE_WHITESPACE** - set the default token value for white space preceeding markup within the context of the root element to an explicit white space value. Use this value in conjunction with the special xml:space attribute to determine how such white space gets classified.
- **GXLHXEC_FEAT_VALIDATE** - perform validation while parsing. See "Usage notes" on page 134 for details of parsing with validation.
- **GXLHXEC_FEAT_SCHEMA_DISCOVERY** - report schema location information and allow for an OSR to be loaded once the information has been reported. GXLHXEC_FEAT_VALIDATE must also be enabled, otherwise gxlpInit will return an error. See "Usage notes" on page 79 for more information on schema discovery. Default: off.
- **GXLHXEC_FEAT_XDBX_INPUT** - indicates that the data presented to z/OS XML in the input buffer is in XDBX binary XML form, rather than conventional text. This feature requires that GXLHXEC_FEAT_VALIDATE is also set, and that the encoding specified in the CCSID parameter is UTF-8. See "Usage notes" on page 134 for more information on XDBX input streams. Default: off.

Note: By using the values of off (zero), W3C XML compliant output is generated. Turning on options GXLHXEC_FEAT_STRIP_COMMENTS and GXLHXEC_FEAT_CDATA_AS_CHARDATA will cause the output to vary from standard compliance.
If none of the features are required, pass the name of a fullword field containing zero. Do not construct a parameter list with a zero pointer in it.

sys_svc_vector
Supplied parameter

Type: GXLHXSV

The name of a structure containing a count of entries that follow and then a list of 31 (64) bit pointers to system service routines. The GXLHXSV member XSV_COUNT must have a value of 0 if no services are provided. For more details on usage, see "Usage notes" on page 75. For more information on exit routines, see the Chapter 8, "z/OS XML System Services exit interface," on page 145 chapter.

sys_svc_parm
Supplied parameter

Type: void *

The name of the area which is passed to all system service exits. This provides for communication between the z/OS XML parser caller and its exit routines. Specify the name of a location containing 0 if no parameter is required for communication.

rc_p

Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

```
#include <stdlib.h>
#include <gxlhxec.h>

void *  pima_p;
long    pima_l;
GXLHXSV sysServiceVec;
int     rc, rsn;

if (pima_p = malloc(GXLHXEC_MIN_PIMA_SIZE))
  { /* pima malloc succeeded */
   pima_l = GXLHXEC_MIN_PIMA_SIZE;
   sysServiceVec.XSV_COUNT = 0;

gxlpInit(pima_p, pima_l,
      GXLHXEC_ENC_UTF_8,
      GXLHXEC_FEAT_STRIP_COMMENTS,
      sysServiceVec,
      NULL,
      &rc, &rsn);

  } /* pima malloc succeeded */
```

Usage notes

System service exit routines cannot get control in the C/C++ environment. Instead, they must be coded to the assembler interface.

Addresses passed in the *system_service_vec* parameter must point to the entry point of the exit being supplied. To obtain the entry point address of a function in 31-bit NOXPLINK DLL compiled module, refer to the FDCB structure in *z/OS Language Environment Vendor Interfaces, SA22-7568*. Otherwise, taking the address of the function will return the entry point address.

This callable service is a direct map to the callable service GXL1INI (GXL4INI). Refer to “Usage notes” on page 134 of GXL1INI (GXL4INI) for additional usage information.

gxlpLoad — load a z/OS XML function

Description

Load a module that implements a z/OS XML function into storage.

Performance Implications

There are no performance implications.

Syntax

```
int gxlpLoad (int function_code,
                   void * function_data,
                   int * rc_p,
                   int * rsn_p)
```

Parameters

`function_code`

Supplied parameter

Type: int

This parameter identifies the z/OS XML function to load. It is the name of an integer value representing the following function:

XEC_LOD_VPARSE

The validating parse function

See gxlhxec.h for the list of function code constants.

`function_data`

Returned parameter

Type: void *

Specify a word of zeroes for this parameter.

`rc_p`

Returned parameter

Type: int *

The name of the area where the service stores the return code.

`rsn_p`

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

None.

Usage notes

This load step is not required for performing non-validating parsing. This operation is only required when using the validating parser. The caller does have the option of loading the load module for the specified function without using this service - either through the z/OS LOAD macro (assembler interface), or by putting it in LPA or the extended LPA. Both the LOAD macro and calls to this service are not allowed when running in an SRB. The use of either interface must be performed in the task before entering SRB mode.

If the required z/OS XML function is made available, either by LOADing the executable load module for it or putting the load module in LPA, this service is not required. Documentation on the LOAD macro can be found in z/OS MVS Programming: Assembler Services Reference, Volume 2, and information on how to load modules into LPA can be found in z/OS Initialization and Tuning Guide.

The load module associated with the function is as follows:

Table 25. Load module for C/C++ parser

Function code	Function performed	Load module name
XEC_LOD_VPARSE	Validating parser function	GXLIMODV

There is no unload service to perform the converse of this function, and none of the other z/OS XML System Services cause the z/OS XML parser to be unloaded. The z/OS XML parser load module will remain in the caller's address space even if the parser is terminated or reset. If multiple parse requests are to be performed in the same address space, make sure to load the z/OS XML parser only once, regardless of whether those parse requests are performed using the same parse instance (PIMA) or not.

gxlpParse — parse a buffer of XML text

Description

The gxlpParse callable service parses a buffer of XML text and places the result in an output buffer.

Performance Implications

Ideal performance will be obtained when the PIMA is sufficiently large to contain all the needed data structures, and the input and output buffers are large enough to process the entire XML document. During the parsing process, the z/OS XML parser constructs persistent information in the PIMA that can be reused within a parse instance. If the caller is going to process multiple documents that contain similar sets of symbols (namespaces and local element and attribute names in particular), then reusing the PIMA will improve performance during the processing of subsequent documents. If this behavior is not required, the PIMA should be cleaned up by calling the termination service and reinitialized by calling the initialization service before using the PIMA for another parse request.

Syntax

```
int gxlpParse(void * PIMA,
              int * option_flags,
              void ** input_buffer_addr,
              long * input_buffer_bytes_left,
              void ** output_buffer_addr,
              long * output_buffer_bytes_left,
              int * rc_p,
              int * rsn_p);
```

Parameters

PIMA

Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA which has been previously initialized with a call to the initialization service.

option_flags

Supplied parameter

Type: int *

This parameter must point to a word with the value 0.

input_buffer_addr

Supplied and returned parameter

Type: void **

The name of the area that contains the address of the buffer with the XML text to parse. The z/OS XML parser updates this parameter to provide important return information when control returns to the caller. See the “Usage notes” on page 137 for details.

input_buffer_bytes_left

Supplied and returned parameter

Type: long *

The name of the area that contains the number of bytes in the input buffer that have not yet been processed. The z/OS XML parser updates this parameter to provide important return information when control returns to the caller. See the “Usage notes” on page 137 for details.

output_buffer_addr

Supplied and returned parameter

Type: void **

The name of the area that contains the address of the buffer where the z/OS XML parser should place the parsed data stream. The z/OS XML parser updates this parameter to provide important return information when control returns to the caller. See the “Usage notes” on page 137 for details.

output_buffer_bytes_left
Supplied and returned parameter

Type: long *

The name of the area that contains the number of available bytes in the output buffer. When the z/OS XML parser returns control to the caller, this parameter will be updated to indicate the number of unused bytes in the output buffer. This buffer must always contain at least a minimum number of bytes as defined by the **GXLHXEC_MIN_OUTBUF_SIZE** constant, declared in header file gxlhxec.h. This service will validate the length of this area against this minimum length value.

rc_p
Returned parameter

Type: int *

The name of the area where the service stores the return code.

rsn_p
Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see "gxlhxr.h (GXLYXR) - defines the return codes and reason codes" on page 218). For reason code descriptions, also see Appendix B, "Reason codes listed by value," on page 161.

Example

```
void * PIMA;

int * option_flags;

void * input_buffer_addr; long input_buffer_bytes_left;

void * output_buffer_addr; long output_buffer_bytes_left;

int rc, rsn;

gxlpParse(PIMA,
          option_flags,
          &input_buffer_addr, &input_buffer_bytes_left,
          &output_buffer_addr, &output_buffer_bytes_left,
          &rc, &rsn);
```

Usage notes

This callable service is a direct map to GXL1PRS (GXL4PRS). Refer to "Usage notes" on page 137 of GXL1PRS (GXL4PRS) for usage information.

gxlpQuery — query an XML document

Description

This service allows a caller to obtain the XML characteristics of a document. The XML characteristics are either the default values, the values contained in an XML declaration or a combination of both.

Performance Implications

There are no performance implications.

Syntax

```
int gxlpQuery (void * work_area,
              long work_area_length,
              void * input_buffer,
              long input_buffer_length,
              GXLHQXD ** return_data,
              int * rc_p,
              int * rsn_p);
```

Parameters

work_area
Supplied parameter

Type: void *

The name of a work area. The work area must be aligned on a doubleword boundary. If not on a doubleword boundary, results are unpredictable. See the “Usage notes” on page 140 for additional details on the use of this area.

work_area_length
Supplied parameter

Type: long

The name of an area containing the length of the work area. The minimum length of this area is declared as a constant **GXLHXEC_MIN_QXDWORK_SIZE** in header file gxlhxec.h . This service validates the length of this area against this minimum length value.

input_buffer
Supplied parameter

Type: void *

The name of an input buffer containing the beginning of the XML document to process. See the “Usage notes” on page 140 for details.

input_buffer_length
Supplied parameter

Type: long

The name of an area containing the length of the input buffer.

return_data
Returned parameter

Type: GXLHQXD **

The pointer to where the service will return the address of the data which describes the XML document characteristics. This return information will contain values that are either extracted from the XML declaration or defaulted according to the XML standard. This return area is mapped by the header file gxlhqxd.h (see “gxlhqxd.h (GXLYQXD) - mapping of the output from the query XML declaration service” on page 216), and is located within the work area specified by the work_area parameter. The caller must not free the work_area until it is done referencing the data returned from this service.

`rc_p`

Returned parameter

Type: int *

The name of the area where the service stores the return code.

`rsn_p`

Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

```
void * work_area;
long work_area_length = XEC_MEM_QIMA_SIZE;
void * input_buffer;
long input_buffer_length;
GXLHQXD * return_data;
int rc, rsn;
gxlpQuery(work_area, work_area_length, input_buffer,
         input_buffer_length, &return_data, &rc, &rsn );
```

Usage notes

This callable service is a direct map to GXL1QXD (GXL4QXD). Refer to “Usage notes” on page 140 of GXL1QXD (GXL4QXD) for usage information.

gxlpTerminate — terminate a parse instance

Description

The gxlpTerminate callable service releases all resources obtained (including storage) by the z/OS XML parser and resets the PIMA so that it can be re-initialized or freed.

Performance Implications

There are no performance implications.

Syntax

```
int gxlpTerminate (void * PIMA,
                   int * rc,
                   int * rsn);
```

Parameters

`PIMA`
Supplied parameter

Type: void *

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

`rc` Returned parameter

Type: int *

The name of the area where the service stores the return code.

`rsn`
Returned parameter

Type: int *

The name of the area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

```
void * PIMA;

int rc, rsn;

gxlpTerminate (PIMA, &rc, &rsn);
```

Usage notes

This callable service is a direct map to GXL1TRM (GXL4TRM). Refer to “Usage notes” on page 142 of GXL1TRM (GXL4TRM) for usage information.

OSR generator API

gxluInitOSRG — initialize an OSR generator instance

Description

Initialize an OSR generator instance. This establishes a context within which the OSR generator performs operations on schemas, Optimized Schema Representations (OSRs), and StringID tables. This context is defined by the OSR generator Instance Memory Area (OIMA).

Performance Implications

The OIMA must be initialized before any OSR generation operations are performed. If operations are to be performed on different OSRs, the caller may enhance performance by resetting the OIMA through a control operation (see gxluControlOSRG), rather than terminating the generator instance and re-initializing. There are implications for memory consumption that must be considered when multiple OSRs are created from the same generator instance. See the usage notes below.

Syntax

```
int gxluInitOSRG (void * oima_p,
                  unsigned long oima_l,
                  int feature_flags,
                  void * sys_svc_parm_p,
                  int * rc_p,
                  int * rsn_p)
```

Parameters

oima_p

Supplied and returned parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA). This area must be at least GXLHXEC_MIN_OIMA_SIZE bytes long. It is used as the work area for the OSR generator.

oima_l

Supplied parameter

Type: unsigned long

The length of the OSR generator Instance Memory Area (OIMA) pointed to by the oima_p parameter.

feature_flags

Supplied parameter

Type: int

The name of the area that contains an integer value representing the OSR generator feature.

sys_svc_parm_p

Supplied parameter

Type: void *

A pointer to an area which is passed to all system service exits, handlers, and resolvers. This provides for communication between the caller of the z/OS XML OSR generator and its exit routines. Specify the NULL pointer if no parameter is required for communication.

rc_p

Returned parameter

Type: int *

A pointer to an area where the service stores the return code.

rsn_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this utility is return code (see below).

Return and Reason Codes:

On return from a call to this utility, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see "gxlhxr.h (GXLYXR) - defines the return codes and reason codes" on page 218). For reason code descriptions, also see Appendix B, "Reason codes listed by value," on page 161.

Example

```
#include <stdlib.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
int           rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
  {       /* oima malloc succeeded */
    oima_l = GXLHXEC_MIN_OIMA_SIZE;

     gxluInitOSRG(oima_p, oima_l,
           0,
                  (void *)handler_parms,
                  &rc, &rsn);

   /* Use the OSR generation instance to perform schema operations ... */
  }       /* oima malloc succeeded */
```

Usage notes

When creating multiple OSRs, the best practice will usually be to initialize one generator instance, and use it for all of the generation operations, with control requests to reset the generator between OSRs. This will consume fewer CPU cycles, and provide better overall performance than initializing and terminating a generator instance for each OSR being created or operated upon. However, all generated OSRs will remain in memory for the duration of the generator instance. If memory constraints are a concern, or you plan to generate OSRs for either a large number of schemas, or for schemas that are very large, you may need to terminate and re-initialize the OSR generator.

gxluControlOSRG — perform an OSR generator control operation

Description

This is a general purpose utility which provides operations for controlling the z/OS XML OSR generator. The operation performed is selected by setting the ctl_option parameter using the constants defined in gxlhxoc.h and gxlhxec.h. These functions include:

GXLHXEC_OSR_CTL_FIN
The caller has finished working with a particular OSR. Reset the necessary structures so that the OIMA can be reused for subsequent generator operations on a different OSR. Receive extended diagnostic information about the current context of the OSR generator.

GXLHXEC_OSR_CTL_DIAG
The caller has finished working with a particular OSR. Receive extended diagnostic information about the current context of the OSR generator.

Performance Implications

The finish-and-reset function allows the caller to re-initialize the OIMA to make it ready to handle a new OSR. This re-initialization path enables the z/OS XML OSR generator to avoid one-time initialization pathlength that’s performed by the initialization service.

Syntax

```
int gxluControlOSRG(void * oima_p,
                  int ctl_operation,
                  void * ctl_data_p,
                  int *  rc_p,
                  int * rsn_p)
```

Parameters

oima_p
Supplied parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA).

ctl_operation
Supplied parameter

Type: int

The name of the parameter containing an integer value representing one of the following operations:

GXLHXEC_OSR_CTL_FIN
This indicates that the caller wants to end processing on the current OSR. The OIMA is re-initialized to allow it to be used to process a new, different OSR. This operation will also return the extended diagnostic information area that is mapped by the gxlhosrd.h header. This includes problem determination information relevant to the current context of the OSR generator.

GXLHXEC_OSR_CTL_DIAG
This indicates that the caller wants to end processing on the current OSR. This operation will return the extended diagnostic information area that is mapped by the gxlhosrd.h header. This includes problem determination information relevant to the current context of the OSR generator.

`ctl_data_p`
Supplied and returned parameter

Type: void *

A pointer to an area that will be used for a purpose that depends on the control operation being performed:

GXLHXEC_OSR_CTL_FIN
A pointer to an area that will receive the address of the extended diagnostic area mapped by gxlhosrd.h. If NULL is specified for this parameter, no extended diagnostic information will be returned. See the usage notes for more about how to use this area.

GXLHXEC_OSR_CTL_DIAG
A pointer to an area that will receive the address of the extended diagnostic area mapped by gxlhosrd.h. If NULL is specified for this parameter, no extended diagnostic information will be returned. See the usage notes for more about how to use this area.

`rc_p`
Returned parameter

Type: int *

A pointer to an area where the service stores the return code.

`rsn_p`
Returned parameter

Type: int *

A pointer to an area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this service is return code (see below).

Return and Reason Codes:

On return from a call to this utility, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) -

defines the return codes and reason codes" on page 218). For reason code descriptions, also see Appendix B, "Reason codes listed by value," on page 161.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
GXLHOSRD *    XDArea_p;
int           rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
   { /* oima malloc succeeded   */
     oima_l = GXLHXEC_MIN_OIMA_SIZE;

     gxluInitOSRG(oima_p, oima_l,
                  0,
                  (void *)handler_parms,
                  &rc, &rsn);
   } /* oima malloc succeeded   */

/* Now perform operations using the generator instance. */

if ((oima_p > 0) && (rc == GXLHXRC_SUCCESS))

  { /* generator ininitialized */

/* generate or load an OSR, generate a StringID table, etc */

   gxluControlOSRG(oima_p,
             GXLHXEC_OSR_CTL_FIN,
             (void *)&XDArea_p,
             &rc, &rsn);

if (rc == GXLHXRC_SUCCESS)
    { /* reset succeeded    */

if (XDArea_p->strID_RC != 0)
     {  /* StringID exit failure   */
      fprintf(stderr,"StringID exit failure: %08x\n",
              XDArea_p->strID_RC);
      ...
      } /* StringID exit failure   */

     } /* reset succeeded         */

    ...

  } /* generator ininitialized */
```

Usage notes

The purpose of the finish-and-reset operation of this service is to reset the necessary structures and fields within the OIMA to prepare the generator instance for reuse without the overhead of full initialization. This reset operation uses fewer CPU cycles than terminating and re-initializing from scratch. However, all schemas that are loaded, and all OSRs and StringID tables that are generated, remain in memory for the duration of the OSR generation instance. If you have a large number of schemas to process, or if the schemas are very large in size, memory constraints may become an issue. In this case, it will be necessary to terminate and re-initialize the OSR generator instance.

The extended diagnostic area returned by the GXLHXEC_OSR_CTL_FIN and GXLHXEC_OSR_CTL_DIAG operations are mapped by gxlhosrd.h. The structure in this header contains assorted diagnostic information about the particular phase of OSR generation that may have failed. The fields of this structure may be used for the duration of the OSR generator instance, but must not be referenced after the instance is terminated. Doing so may result in unpredictable results.

gxluTermOSRG — terminate an OSR generator instance

Description

The gxlpTermOSRG utility releases all resources obtained by the z/OS XML OSR generator. It also sets the eyecatcher in the OIMA to prevent it from being reused by other OSR API functions, with the exception of re-initialilization by gxluInitOSRG.

Performance Implications

There are no performance implications.

Syntax

```
int gxluTermOSRG(void * oima_p,
                 int *  rc_p,
                 int * rsn_p)
```

Parameters

oima_p

Supplied parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA).

rc_p

Returned parameter

Type: int *

A pointer to an area where the service stores the return code.

rsn_p

Returned parameter

Type: int *

A pointer to an area where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this service is return code (see below).

Return and Reason Codes:

On return from a call to this utility, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
int           rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
      { /* oima malloc succeeded   */
        oima_l = GXLHXEC_MIN_OIMA_SIZE;

        gxluInitOSRG(oima_p, oima_l,
                     0,
                     (void *)handler_parms,
                     &rc, &rsn);
    }  /* oima malloc succeeded   */

/* Now perform operations using the generator instance. */
if ((oima_p > 0) && (rc == GXLHXRC_SUCCESS))
    {  /* generator ininitialized */

    /* generate or load an OSR, generate a StringID table, etc */

    gxluTermOSRG(oima_p,
                 &rc, &rsn);
  /* Do not use any resources that the OSR generator   */
  /* has allocated from here on.                       */

  }  /* generator ininitialized */
```

Usage notes

This utility does not free the OSR Generator Instance Memory Area (OIMA). It is up to the caller to free the OIMA after termination completes. gxluTermOSRG will, however, free any binary OSR buffers, StringID tables, and extended diagnostic areas that may have been allocated during the OSR generator instance. Once termination has completed, you must not reference any of these areas, or any extended diagnostic areas that may have been created during the generator instance. It is the caller's responsibility to create persistent copies of these structures as needed while the generator instance is active.

gxluLoadSchema — load a schema into the OSR generator

Description

This utility is used to load text schemas into the OSR generator. It is called once for each schema that will be processed to create an Optimized Schema Representation.

Performance Implications

There are no performance implications.

Syntax

```
int gxluLoadSchema(void * oima_p,
                   char * schema_resource_p,
                   int *  rc_p,
                   int * rsn_p)
```

Parameters

oima_p

Supplied parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA).

schema_resource_p

Supplied parameter

Type: char *

A pointer to the schema resource to process. This parameter must contain a NULL terminated, IBM-1047 text string representing one of the following:

- The pathname of a file in the z/OS UNIX file system containing the schema in text form.
- URI specifying the location of the schema text to load. URIs are indicated by a scheme name, followed by a colon, followed by a relative URI reference. See RFC 3986 (http://tools.ietf.org/html/rfc3986) for a complete description of URIs.

Whether the resource passed is a URI or a pathname to a file, the name must represent an absolute path. Relative paths cannot be processed.

rc_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the return code.

rsn_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this service is return code (see below).

Return and Reason Codes:

On return from a call to this utility, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
char   schema_uri[URI_LEN] = "file:///u/user01/myschema.xsd";
int           rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
      { /* oima malloc succeeded   */
         oima_l = GXLHXEC_MIN_OIMA_SIZE;

        gxluInitOSRG(oima_p, oima_l,
                      0,
                     (void *)handler_parms,
                      &rc, &rsn);
    }  /* oima malloc succeeded   */

/* Now perform operations using the generator instance. */
if ((oima_p > 0) && (rc == GXLHXRC_SUCCESS))
    {  /* generator initialized */

       gxluLoadSchema(oima_p,
                      schema_uri,
                      &rc, &rsn);

        if (rc == GXLHXRC_SUCCESS)
        {/* schema load succeeded   */

 /*generate an OSR from the loaded schema*/


    ...
    }  /* schema load succeeded   */

   ...
   } /* generator initialized */
```

Usage notes

Call this service iteratively to load one or more schemas that will be processed to create an OSR. Once a schema has been loaded, the schema text buffer specified by the schema_resource_p parameter may be re-used for other purposes.

gxluSetStrIDHandler — specify the StringID handler for OSR generation

Description

This utility allows the caller to specify a StringID handler service to the OSR generator. The StringID handler utility allows the caller to avoid making StringID calls at parse time for a number of symbols. This handler must be written in C.

Performance Implications

There are no performance implications.

Syntax

```
int gxluSetStrIDHandler(void * oima_p,
                  char * dll_name_p,
                  char * func_name_p,
                  int *  rc_p,
                  int * rsn_p)
```

Parameters

oima_p
Supplied parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA).

dll_name_p
Supplied parameter

Type: char *

A pointer to the NULL terminated name of the DLL containing the StringID handler executable. This string must be in the IBM-1047 code page. A NULL string indicates that the current StringID handler should be unset, and StringIDs no longer used during the creation of OSRs.

func_name_p
Supplied parameter

Type: char *

A pointer to the NULL terminated name of the StringID handler within the DLL. This string must be in the IBM-1047 code page. If the dll_name_p parameter above is NULL, this function name is ignored.

rc_p
Returned parameter

Type: int *

A pointer to an area where the utility stores the return code.

rsn_p
Returned parameter

Type: int *

A pointer to an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this service is return code (see below).

Return and Reason Codes:

On return from a call to this utility, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
char   dll_name[SIZE]  = "dllpath/dllname.so";
char   func_name[SIZE] = "strIDHandler";
int           rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
      { /* oima malloc succeeded   */
      oima_l = GXLHXEC_MIN_OIMA_SIZE;

 gxluInitOSRG(oima_p, oima_l,
         0,
         (void *)handler_parms,
         &rc, &rsn);
    }  /* oima malloc succeeded   */

/* Now set a StringID handler that will be used to  */
/* create StringIDs when OSRs are generated.        */

if ((oima_p > 0) && (rc == GXLHXRC_SUCCESS))
    {  /* generator initialized */
    gxluSetStrIDHandler (oima_p,
              dll_name, func_name,
              &rc, &rsn);

    if (rc == GXLHXRC_SUCCESS)
       { /* set handler succeeded   */

       <continue processing using the StringID handler>

         ...
       }  /* set handler succeeded   */

      ...
      } /* generator initialized */
```

Usage notes

This handler differs from the other handlers and resolvers provided to the OSR generator in that it must be written in C. Both the validating z/OS XML parser and the OSR generator allow the caller to specify a StringID handler, and by implementing this handler as a C DLL, the same source may be used in both environments. A key difference is that this handler must be compiled and linked with conventional C and Language Environment capabilities for the OSR generator environment, while it must be built using Metal C for the parser.

The DLL containing the StringID handler will be loaded in order to obtain a function pointer to it. The function pointer will be kept within the OIMA until a

StringID is needed during OSR generation. The DLL path must reside in one of the paths specified in the LIBPATH environment variable.

This routine may be called more than once during an OSR generation instance to change the StringID handler that the generator uses.

gxluSetEntityResolver — specify the entity resolver for OSR generation

Description

This utility allows the caller to specify an entity resolver to the OSR generator. This resolver must be written in Java.

Performance Implications

There are no performance implications.

Syntax

```
int gxluSetEntityResolver(void * oima_p,
                   char * class_name_p,
                   int *  rc_p,
                   int * rsn_p)
```

Parameters

oima_p
Supplied parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA).

class_name_p
Supplied parameter

Type: char *

A pointer to the NULL terminated name of a Java class that implements the XMLEntityResolver interface of the XML4J parser (see the usage notes below). This string must be in the IBM-1047 code page. A NULL string indicates that the current entity resolver should be unset, and the default resolver used during the creation of the OSR.

rc_p
Returned parameter

Type: int *

A pointer to an area where the utility stores the return code.

rsn_p
Returned parameter

Type: int *

A pointer to an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this service is return code (see below).

Return and Reason Codes:

On return from a call to this utility, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see "gxlhxr.h (GXLYXR) - defines the return codes and reason codes" on page 218). For reason code descriptions, also see Appendix B, "Reason codes listed by value," on page 161.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
char          class_name[SIZE]  = "xml/appl/handlers/EntityResolver";
int           rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
      { /* oima malloc succeeded   */
      oima_l = GXLHXEC_MIN_OIMA_SIZE;

 gxluInitOSRG(oima_p, oima_l,
         0,
         (void *)handler_parms,
         &rc, &rsn);
    }  /* oima malloc succeeded   */

/* Now set an entity resolver that will be used during  */
/* OSR generation.                                      */

if ((oima_p > 0) && (rc == GXLHXRC_SUCCESS))
    {  /* generator initialized */
    gxluSetEntityResolver(oima_p,
                 class_name,
                 &rc, &rsn);

    if (rc == GXLHXRC_SUCCESS)
       { /* set resolver succeeded   */

       <continue processing using the entity resolver>

         ...
       }  /* set resolver succeeded   */

      ...
      } /* generator initialized */
```

Usage notes

Although this is a C interface, the entity resolver must be implemented in Java. This resolver will be provided to the XML4J parser, which is used during the OSR generation process. The resolver must implement the XMLEntityResolver interface of the Xerces Native Interface (XNI), including the return of an XMLInputSource object. See the XMLEntityResolver documentation at http://xerces.apache.org/xerces2-j/javadocs/xni/index.html.

This routine may be called more than once during an OSR generation instance to change the entity resolver that the generator uses.

gxluLoadOSR — load an OSR into the OSR generator

Description

This utility is used to load an Optimized Schema Representation into the OSR generator. Once loaded, the OSR may be processed using one of the OSR generator operations.

Performance Implications

There are no performance implications.

Syntax

```
int gxluLoadOSR(void * oima_p,
                void * osr_p,
                int osr_l,
                int *  rc_p,
                int * rsn_p)
```

Parameters

oima_p

Supplied parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA).

osr_p

Supplied parameter

Type: void *

A pointer to a buffer containing an OSR.

osr_l

Supplied parameter

Type: int

The length of a buffer containing an OSR.

rc_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the return code.

rsn_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this service is return code (see below).

Return and Reason Codes:

On return from a call to this utility, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
char          osrbuf[OSR_BUFFER_LEN];
int           osrbuf_l;
int           rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
      { /* oima malloc succeeded   */
      oima_l = GXLHXEC_MIN_OIMA_SIZE;

 gxluInitOSRG(oima_p, oima_l,
         0,
         (void *)handler_parms,
         &rc, &rsn);
    }  /* oima malloc succeeded   */

<acquire the OSR from a persistent location like a file ...>
/* Load an OSR to be processed.    */

if ((oima_p > 0) && (rc == GXLHXRC_SUCCESS))
    {  /* generator initialized */
    gxluLoadOSR(oima_p,
               (void *)osrbuf
                osrbuf_l,
                &rc, &rsn);

    if (rc == GXLHXRC_SUCCESS)
       { /* OSR load succeeded   */

       <process the loaded OSR>

         ...
       }  /* OSR load succeeded   */

      ...
      } /* generator initialized */
```

Usage notes

Use this utility when you need to query an OSR that has already been created from one or more human-readable schemas. This is useful, for instance, when a caller needs access to a StringID table from an existing OSR. This allows the StringID table to be used by the validating parser at parse time.

gxluGenOSR — generate an Optimized Schema Representation (OSR)

Description

This utility generates an optimized representation of one or more XML schemas.

Performance Implications

There are no performance implications.

Syntax

```
unsigned int gxluGenOSR(void * oima_p,
                void ** schema_osr_p_p,
                int * rc_p,
                int * rsn_p)
```

Parameters

oima_p
Supplied parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA).

schema_osr_p_p
Returned parameter

Type: void **

A pointer to an area to receive the address of the optimized schema representation generated by this utility. See the usage notes below for important details about this parameter.

rc_p
Returned parameter

Type: int *

A pointer to an area where the utility stores the return code.

rsn_p
Returned parameter

Type: int *

A pointer to an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this utility is the length of the OSR buffer returned to the caller through the schema_osr_p parameter. If there is a problem during the generation of the OSR, the value returned will be zero. See the usage notes below for more information about this value and the OSR buffer returned.

Return and Reason Codes:

Register 15 will contain the return value of this utility. The return and reason code are both set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
char          schema_uri[URI_LEN] = "file:///u/user01/myschema.xsd";
void *        osr_p;
int          osr_l;
int          rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
      { /* oima malloc succeeded   */
      oima_l = GXLHXEC_MIN_OIMA_SIZE;

 gxluInitOSRG(oima_p, oima_l,
         0,
         (void *)handler_parms,
         &rc, &rsn);
    }  /* oima malloc succeeded   */


/* Load a schema and create an OSR from it. */

if ((oima_p > 0) && (rc == GXLHXRC_SUCCESS))
    {  /* generator initialized */
    gxluLoadSchema(oima_p,
               schema_uri,
               &rc, &rsn);

    if (rc == GXLHXRC_SUCCESS)
       { /* schema load succeeded   */

       /* Generate the OSR */
       osr_l = gxluGenOSR(oima_p,
                          &osr_p,
                          &rc, &rsn);

       if (osr_l > 0) then
        {  /* OSR generate succeeded  */

   <write the OSR out to a persistent repository>
   <like a file or a database so that it can be>
   <used later for parsing a document>

           ...
           }      /* OSR generate succeeded  */
       ...
       }  /* schema load succeeded   */

      ...
      } /* generator initialized */
```

Usage notes

This utility generates Optimized Schema Representations in a manner similar to the xsdosrg command (see Appendix C, "xsdosrg command reference," on page 213). It provides additional flexibility and control by allowing the caller to use the following handlers to augment the default generator behavior:

StringID handler
: This handler generates and/or returns an integer identifier that serves as a handle for a given string. These strings are most often the components of qualified names that are encountered in the schema text during processing. This must be implemented as a C routine, and built for the C Language Environment. If no StringID handler is specified, then StringIDs will not be used during the generation of the OSR. All qualified names and other strings for which IDs could be used will instead be present in the OSR in their text form. The same handler may be used by the validating parser when built for the Metal C environment.

entity resolver
: A Java routine that receives control when a reference to an external entity is made from one schema to another through an include, import, or redefine XML Schema construct. It acquires the external schema from an appropriate source, and returns it to the OSR generator for further processing. If no entity resolver is specified, the default entity resolver from the XML4J parser is used.

One or both of these routines may be specified to the OSR generator through the gxluSetStringID ("gxluSetStrIDHandler — specify the StringID handler for OSR generation" on page 91) and gxluSetEntityResolver ("gxluSetEntityResolver — specify the entity resolver for OSR generation" on page 94) utilities. Once set, the generator will make use of them until they are changed to a different value.

This utility will allocate the buffer used to receive the generated OSR, and will return the length of the buffer as its return value. The maximum length of an OSR that will be returned is 2 GB. The buffer remains allocated for the duration of the OSR generator instance, and gets freed when the instance is terminated. The caller may use or copy the OSR to another location as long as the instance is active. Referencing the OSR buffer after the generator instance has been terminated may result in unpredictable results. This buffer may also be written to a permanent location, such as a z/OS UNIX file or an MVS data set, so that it can be used again at some point in the future.

gxluGenStrIDTable — generate StringID table from an OSR

Description

This utility will extract generate and return the StringID table associated with the current OSR for this generator instance. See the usage notes for a description of how to make an OSR current.

Performance Implications

There are no performance implications.

Syntax

```
int gxluGenStrIDTable(void * oima_p,
                 GXLHXSTR ** strid_tbl_p_p,
                 int * rc_p,
                 int * rsn_p)
```

Parameters

oima_p

Supplied parameter

Type: void *

A pointer to an OSR generator Instance Memory Area (OIMA).

strid_tbl_p_p

Supplied and returned parameter

Type: GXLHXSTR **

A pointer to an area that will receive the address of a table of containing the StringIDs that are generated from the current OSR. See the usage notes below for more details.

rc_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the return code.

rsn_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this utility is the length of the StringID table returned to the caller through the strid_tbl_p_p parameter. If StringIDs were not in use when the current OSR was originally generated, the return value will be zero, and the pointer specified by strid_tbl_p will remain unchanged. If there is a problem during the generation of the StringID table, the value returned will be -1. See the usage notes below for more information about this value, and the StringID table returned.

Return and Reason Codes:

Register 15 will contain the return value of this utility (see above). The return and reason code are both set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see "gxlhxr.h (GXLYXR) - defines the return codes and reason codes" on page 218). For reason code descriptions, also see Appendix B, "Reason codes listed by value," on page 161.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gxlhosrg.h>
#include <gxlhxec.h>

void *        oima_p;
unsigned long oima_l;
char          handler_parms[128];
char          osrbuf[OSR_BUFFER_LEN];
int           osrbuf_l;
GXLHXSTR *    strIDTbl_p;
int           strIDTbl_l;
int           osr_l;
int           rc, rsn;

if (oima_p = malloc(GXLHXEC_MIN_OIMA_SIZE))
      { /* oima malloc succeeded   */
      oima_l = GXLHXEC_MIN_OIMA_SIZE;

 gxluInitOSRG(oima_p, oima_l,
         0,
         (void *)handler_parms,
         &rc, &rsn);
    }  /* oima malloc succeeded   */

<acquire the OSR from a persistent location like a file>

/* Load the OSR to operate on. */

if ((oima_p > 0) && (rc == GXLHXRC_SUCCESS))
    {  /* generator initialized */
    gxluLoadOSR(oima_p,
               (void *)osrbuf,
                osrbuf_l,,
                &rc, &rsn);

    if (rc == GXLHXRC_SUCCESS)
       { /* OSR load succeeded   */

       /* Generate the OSR */
       strIDTbl_l = gxluGenSTRIDTable(oima_p,
                          &strIDTbl_p,
                          &rc, &rsn);

       if (strIDTbl_l > 0) then
        { /* strID table generated   */

   <write the StringID table out to a persistent>
   <repository like a file or a database so that>
   <it can be used later when parsing a document>

           ...
           }      /* strID table generated  */
       ...
       }  /* OSR load succeeded      */

      ...
      } /* generator initialized */
```

Usage notes

The StringID table is generated from the OSR that has been made current through either a gxluGenOSR or a gxluLoadOSR request. The actual length of the StringID table is calculated during table generation, and cannot be known ahead of time.

For this reason, the gxluGenStrIDTable service will return the address and length of the generated table on success. The table remains allocated for the duration of the OSR generator instance, and gets freed when the instance is terminated. The caller may use or copy the StringID table to another location as long as the instance is active. Referencing the StringID table after the generator instance has been terminated may result in unpredictable results.

StringID tables may be generated from OSRs that were created either with or without StringIDs. If no StringIDs were used when the OSR was originally generated, this service will assign the StringID values to return in the table. Callers who wish to control the values of StringIDs must use the StringID handler interface at OSR generation time.

The format of the StringID table that the OSR generator creates is defined by the gxlhxstr.h header file. See the definition of this header file below for more details.

gxluGetStringIDs — generate StringID table from an OSR

Description

This utility will generate and return the StringID table associated with the supplied OSR.

Performance Implications

There are no performance implications.

Syntax

```
GXLHXSTR * gxluGetStringIDs(const void *OSR_p,
                            int *rc_p,
                            int *rsn_p)
```

Parameters

OSR_p

Supplied parameter

Type: void *

A pointer to an OSR.

rc_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the return code.

rsn_p

Returned parameter

Type: int *

A pointer to an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this utility is the address of the generated StringID table. Once this table is no longer needed, it must be freed by a call to gxluFreeStringIDs. If there is a problem during the generation of the StringID table, the value returned will be NULL. See the usage notes for gxluFreeStringIDs for more information about the StringID table returned.

Return and Reason Codes:

The return and reason code are both set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see "gxlhxr.h (GXLYXR) - defines the return codes and reason codes" on page 218). For reason code descriptions, also see Appendix B, "Reason codes listed by value," on page 161.

gxluFreeStringIDs — free a StringID table

Description

This utility will free a StringID table that was returned from a call to gxluGetStringIDs

Performance Implications

There are no performance implications.

Syntax

```
void gxluFreeStringIDs(GXLHXSTR *table_p)
```

Parameters

`table_p`

Supplied parameter

Type: GXLHXSTR *

The StringID table to be freed.

All parameters in the parameter list are required.

Return Value:

There are no return values.

Return and Reason Codes:

There are no return and reason codes.

Usage notes

The StringID table that is to be freed must have been generated by a call to gxluGetStringIDs, and not gxluGenStrIDTable. Attempting to free a string ID table that was not generated by gxluGetStringIDs will have no effect.

gxluGetRootElements — retrieve the root elements from an OSR

Description

This utility allows the caller to query the OSR generator for a set of all possible root elements that may be used with this OSR.

Performance Implications

There are no performance implications.

Syntax

```
const GXLHXRE* gxluGetRootElements(void * osr_p,
                                   int * rc_p,
                                   int * rsn_p)
```

Parameters

osr_p

Supplied parameter

Type: void *

A pointer to the OSR from which information is to be extracted.

rc_p

Returned parameter

Type: int *

The name of the area where the utility stores the return code.

rsn_p

Returned parameter

Type: int *

The name of an area where the utility stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this utility is a pointer to a GXLHXRE structure containing all of the root elements within the OSR. This structure must be freed by gxluFreeRootElements.

Return and Reason Codes:

The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

gxluFreeRootElements — free a root element structure

Description

This utility will free a root element structure that was returned from a call to gxluGetRootElements.

Performance Implications

There are no performance implications.

Syntax

```
void gxluFreeRootElements(GXLHXRE *table_p)
```

Parameters

table_p
Supplied parameter

Type: GXLHXRE *

The root element structure to free.

All parameters in the parameter list are required.

Return Value:

There are no return values.

Return and Reason Codes:

There are no return and reason codes.

gxluGetTargetNamespaces — retrieve the target namespaces from an OSR

Description

This utility allows the caller to query the OSR generator for all target namespaces that are associated with this OSR.

Performance Implications

There are no performance implications.

Syntax

```
const GXLHXTN* gxluGetTargetNamespaces(void * osr_p,
                                       int * rc_p,
                                       int * rsn_p)
```

Parameters

osr_p
Supplied parameter

Type: void *

A pointer to an OSR from which information is to be extracted.

rc_p

Returned parameter

Type: int *

The name of the area where the utility stores the return code.

rsn_p

Returned parameter

Type: int *

The name of an area where the utility stores the reason code .The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return Value:

The value returned by this utility is a pointer to a GXLHXTN structure containing all of the target namespaces associated with the OSR. This structure must be freed by a call to gxluFreeNamespaces. A schema without a target namespace will be represented by a URI in the GXLHXTN structure with 0-length.

Return and Reason Codes:

The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in the header file gxlhxr.h (see "gxlhxr.h (GXLYXR) - defines the return codes and reason codes" on page 218). For reason code descriptions, also see Appendix B, "Reason codes listed by value," on page 161.

Usage notes

If the OSR being queried is an OSR generated from a release prior to z/OS V1R12, the returned list of targetNamespaces will only include namespaces found from the possible root elements.

gxluFreeNamespaces — free a namespace structure

Description

This utility will free a namespace structure that was returned from a call to gxluGetTargetNamespaces.

Performance Implications

There are no performance implications.

Syntax

```
void gxluFreeTargetNamespaces(GXLHXTN *table_p)
```

Parameters

oima_p

Supplied parameter

Type: GXLHXTN *

The namespace structure to be freed.

All parameters in the parameter list are required.

Return Value:

There are no return values.

Return and Reason Codes:

There are no return and reason codes.

GXLPSYM31 (GXLPSYM64) — StringID handler

Description

This handler accepts an input string and performs a lookup for its corresponding symbol, which is identical to the string itself. If the symbol has been located, the exit returns the StringID associated with the symbol. If the string does not have a defined symbol, a symbol is created for the string and a StringID is assigned to it.

Performance Implications

There are no performance implications.

Syntax

```
int gxlpSym31(void ** sys_svc_p,
              char * string_p,
              int string_l,
              unsigned int * string_id_p,
              int ccsid,
              int * handler_diag_p,
              int * rc_p)
```

Parameters

sys_svc_p

Supplied parameter

Type: void **

A pointer to the system service parameter that was passed to the z/OS XML OSR generator at initialization time.

string_p

Supplied parameter

Type: char *

The string to return an ID for. The length of the string is variable, and is specified by the *string_l* parameter.

string_l

Supplied parameter

Type: int

An integer containing the length of the string pointed to by the string parameter.

`string_id_p`
Returned parameter

Type: unsigned int *

A pointer to an integer where the handler stores the numeric identifier for the string. The range of valid values is 1 to 2 GB - 1.

`ccsid`
Supplied parameter

Type: int

The Coded Character Set IDentifier (CCSID) that identifies the character set of the string. The z/OS XML parser will provide the same CCSID in this parameter that the caller of the parser specified at parser initialization time.

`handler_diag_p`
Returned parameter

Type: int *

A pointer to an integer where the handler can store any diagnostic information (usually a reason code). This will be stored in the diagnostic area and made available on the gxluControlOSRG call.

`rc_p`
Returned parameter

Type: int *

A pointer to an integer where the handler can store a return code. A return code value of zero means success; any nonzero return code indicates failure.

Return Codes:

The z/OS XML OSR generator uses the convention that the handler will provide a return code value of zero when successful. Any nonzero value indicates failure. If a nonzero return code is provided by the exit, the z/OS XML OSR generator saves it in the extended diagnostic area so that the caller of the parser has access to it by calling gxluControlOSRG.

Example

None.

Default Implementation

There is no default implementation. If this handler is not specified by the caller, StringIDs are not used by the z/OS XML OSR Generator.

Chapter 7. z/OS XML parser API: Assembler

How to invoke the z/OS XML System Services assembler API

This section provides information on how to invoke the z/OS XML System Services assembler API.

Callers written in assembler can invoke the z/OS XML System Services assembler API by binding the z/OS XML parser's callable service stubs to their module. The callable service stubs can be found in SYS1.CSSLIB. Alternatively, the addresses of the APIs can be obtained from system control blocks. The following is a list of offsets for the callable services first and second tables (all offsets are in hex):

1. +10 — Pointer to CVT (field FLCCVT in IHAPSA)
2. +220 — Pointer to the callable services first table (field CVTCSRT in CVT)
3. +48 — Pointer to the z/OS XML parser callable services second table (entry 19)

   Note: Prior to z/OS V1R7, this field will point to the address of an undefined callable service. In z/OS V1R7 and later releases, this field is zero until the z/OS XML parser initialization routine fills it in. To avoid calling z/OS XML System Services when it is not present, the caller first needs to verify that it is running on V1R7 or later, and that this field in the callable services first table is non-zero.
4. +nn — The offset for each callable service in hex is listed below.

The following stubs are provided for 31- and 64-bit mode callers:

Table 26. Caller stubs and associated offsets

Stub	Second Table offset (hex)
GXL1INI — 31-bit parser initialization	10
GXL1PRS — 31-bit parse	14
GXL1TRM — 31-bit parser termination	18
GXL1CTL — 31-bit parser control operation	1C
GXL1QXD — 31-bit query XML document	20
GXL1LOD — 31-bit load a function	24
GXL4INI — 64-bit parser initialization	28
GXL4PRS — 64-bit parse	30
GXL4TRM — 64-bit parser termination	38
GXL4CTL — 64-bit parser control operation	40
GXL4QXD — 64-bit query XML document	48
GXL4LOD — 64-bit load a function	50

Note: The 64-bit stubs are defined with 8 byte pointers.

Following the offsets to the caller stubs, at offset 78 (hex) from the start of the second table, is an 8 byte field of bits. These bits indicate the presence of a particular z/OS XML capability. Callers may reference these bits to determine if the function or feature that they intend to use is supported by the installed version of z/OS XML.

The following table lists the bits that are defined, along with their descriptions:

Table 27. Capability bits

Capability bit	Description
'0000000000000001'X	XDBX validation is available

The following assembler code is an example of how to call a z/OS XML parser service. The example assumes the caller uses the CVT field names instead of hard coding those offsets.

```
LLGT 15,CVTPTR         R15L -> CVT, R15H = 0
L    15,CVTCSRT-CVT(15) Get the CSRTABLE
L    15,72(15)         Get CSR slot 19 (zero based) for XML parser
L    15,16(15)         Get address of GXL1INI from XML second table.
BALR 14,15             Branch to XML service.
```

z/OS XML parser Assembler API

This section lists the assembler callable services interface used for the z/OS XML parser. The following rules apply to some or all of the callable services listed below:

- The 31- and 64-bit versions of the services were designed to work independently of one another. For example, the following sequence of calls would not work: GXL1INI (31-bit service) followed by GXL4PRS (64-bit service).
- The 31- and 64-bit versions of the services are documented together with any differences for 64- bit shown in parenthesis, after its corresponding 31-bit description.
- In AMODE 31, all address and length parameters of the z/OS XML parser API are 4 bytes long. In AMODE 64, these fields are 8 bytes long.
- In AMODE 31, the parsed data stream produced by the z/OS XML parser contains length fields that are all 31 bits (4 bytes) long. In AMODE 64, the field in the buffer header representing the length of the output buffer used is 64-bits (8 bytes) long, while all record length fields in the data stream are 31-bit (4 byte) values.

API entry points

The z/OS XML parser API contains 5 entry points for each addressing mode (AMODE) type (31- or 64-bit):

- GXL1CTL (GXL4CTL) — perform a parser control operation
- GXL1INI (GXL4INI) — initialize a parse instance
- GXL1PRS (GXL4PRS) — parse an input stream
- GXL1QXD (GXL4QXD) — query an XML document
- GXL1TRM (GXL4TRM) — terminate a parse instance
- GXL1LOD (GXL4LOD) — load a function

Common register conventions

The following sections describe common register conventions used for all of the z/OS XML parser's callable services.

Input registers

When a caller invokes the z/OS XML parser, these registers have the following meaning:

Table 28. Input register conventions

Register	Contents
1	Address of a standard parameter list containing 31 (64) bit addresses.
14	Return address.

Output registers

When the z/OS XML parser returns to the caller, these registers have the following meaning:

Table 29. Output register conventions

Register	Contents
0-1	Unpredictable
2-13	Unchanged
14	Unpredictable
15	Return code (return code is also a parameter)

Table 30. Output access register conventions

Access Register	Contents
0-1	Unpredictable
2-13	Unchanged
14-15	Unpredictable

Environmental requirements

The following are environmental requirements for the caller of any z/OS XML parser service:

Minimum authorization
any state and any PSW key

Dispatchable unit mode
Task or SRB

Note: GXL1LOD (GXL4LOD) can only operate in Task mode.

Cross memory mode
PASN=HASN=SASN or PASN^=HASN^=SASN

AMODE
31-bit (64-bit)

ASC mode
primary

Interrupt status
enabled for I/O and external interrupts

Locks no locks held

Control parameters
Control parameters and all data areas the parameter list points to must be addressable from the current primary address space.

Using the recovery routine

z/OS XML provides an ARR recovery routine to assist with problem determination and diagnostics. This is an optional routine and can be turned on and off as desired. See“ARR recovery routine” on page 158 for more information.

Restriction: When running in either SRB mode or under an existing FRR routine, the ARR recovery routine cannot be used.

GXL1CTL (GXL4CTL) — perform a parser control function

Description

This is a general purpose service which provides control functions for interacting with the z/OS XML parser. The function performed is selected by setting the *ctl_option* parameter using the constants defined in GXLYXEC. These functions include:

XEC_CTL_FIN
The caller has finished parsing the document. Reset the necessary structures so that the PIMA can be reused on a subsequent parse, and return any useful information about the current parse. For more information on this function, see “XEC_CTL_FIN” on page 117.

XEC_CTL_FEAT
The caller wants to change the feature flags. A XEC_CTL_FIN function will be done implicitly.

Note: Some feature flags are not supported on GXL1CTL (GXL4CTL). See “XEC_CTL_FEAT” on page 118 for information on which feature flags are not supported.
For more information on this function, see “XEC_CTL_FEAT” on page 118.

XEC_CTL_LOAD_OSR
The caller wants to load and use an Optimized Schema Representation (OSR) for a validating parse. For more information on this function, see “XEC_CTL_LOAD_OSR” on page 120.

XEC_ CTL_QUERY_MIN_OUTBUF
The caller is requesting the minimum output buffer size required on a subsequent parse. This function will also enable the parse to be continued after a XRSN_BUFFER_OUTBUF_SMALL reason code has been received from GXL1PRS(GXL4PRS).

Note: Finish and reset processing is performed by all operations available through this control service, except XEC_CTL_QUERY_MIN_OUTBUF and XEC_CTL_LOAD_OSR. See the descriptions of these operations under ctl_option for more information.
For more information on this function, see “XEC_CTL_QUERY_MIN_OUTBUF” on page 121.

XEC_CTL_ENTS_AND_REFS
The caller can request additional flexibility when processing character and entity references as follows:

- When an unresolved entity reference is encountered, the caller can request that the parser stop processing and return an error record.
- When a character reference which cannot be represented in the current code page is encountered, z/OS XML System Services places a dash (-)

in the output stream for that character. The caller may specify, with this control call, to output a character other than dash (-) in the output stream.

- When a character reference which cannot be represented in the current code page is encountered, the caller can request, using this control call, an additional output record to be generated in the output stream that contains information about this character reference.

Note:

1. Finish and reset processing is performed for this control operation. See "Usage notes" on page 116 for more information.
2. If the parse instance has been initialized to process XDBX binary XML streams, then the input stream will never have any entity references to resolve. Performing the XEC_CTL_ENTS_AND_REFS operation will have no effect on the output of the parser. In order to prevent accidental attempted use of this operation in this environment, the parser will return a failure.

For more information on this function, see "XEC_CTL_ENTS_AND_REFS" on page 123.

XEC_CTL_LOAD_FRAG_CONTEXT

The caller wants to load fragment context including fragment path and namespace binding information for document fragment parsing.

Note:

1. This control operation does not perform finish and reset processing through the control service. See the description in ctl_option for more information.
2. Fragment parsing is not supported for XDBX input. For this reason, attempting to load a fragment context for parse instances initialized to handle XDBX streams will fail.

For more information on this function, see "XEC_CTL_LOAD_FRAG_CONTEXT" on page 124.

XEC_CTL_FRAGMENT_PARSE

The caller wants to enable or disable document fragment parsing.

Note:

1. This control operation does not perform finish and reset processing through the control service. See the description in ctl_option for more information.
2. Fragment parsing is not supported for XDBX input. For this reason, attempting to enable document fragment parsing for parse instances initialized to handle XDBX streams will fail.

For more information on this function, see "XEC_CTL_FRAGMENT_PARSE" on page 126.

XEC_CTL_RESTRICT_ROOT

The caller can restrict the root element name on the next parse. This operation is only valid if the PIMA has been configured for validation and schema information is requested. For more information on this function, see "XEC_CTL_RESTRICT_ROOT" on page 128.

XEC_CTL_ERROR_HANDLING

With this control operation, the caller can do the following:

- Enable the creation of auxiliary records which can include the location of an error in the XML document, the string which is in error, and also a possible expected string.
- Enable position indexes to be present in the error location path in order to facilitate locating the error.

For more information on this function, see "XEC_CTL_ERROR_HANDLING" on page 129.

Performance Implications

The finish/reset function allows the caller to re-initialize the PIMA to make it ready to handle a new XML document. This re-initialization path enables the z/OS XML parser to preserve its existing symbol table, and avoid other initialization pathlength that's performed by calling GXL1INI (GXL4INI).

Example

For an AMODE 31 example using this callable service, see "GXL1CTL example" on page 221. For an AMODE 64 example using this callable service, see "GXL4CTL example" on page 225.

Usage notes

The purpose of the finish/reset function of the GXL1CTL (GXL4CTL) service is to perform the following:

- Reset the necessary structures and fields within the PIMA to effect a re-initialization so that it can be reused without the overhead of full initialization. See the table below for list of structures and fields reset by each control function.
- Allow the z/OS XML parser to return extended diagnostic information to the caller in the event of a failure. This allows the caller to identify certain problems that can be corrected.
- The "finish and reset" operation can be thought of as the most basic control operation that is a functional subset of all control operations. It resets the state of the parser to the original state immediately after the parse instance was first initialized. This state includes the feature flags. If the caller initializes a parse instance, then changes the feature settings with a feature control operation, and still later performs a "finish and reset" control operation, the feature flags will revert back to those settings at the time the parse instance was originally initialized. If the caller wishes to retain the current feature settings during a parser reset, they should simply perform another feature control operation with the current feature set.
- The OSR load operation allows the caller to specify an OSR for the parser to use, and to bind a handle to associate with to it. The GXLYXOSR macro provides the interface for passing information to the parser about the OSR. See Appendix D, "C/C++ header files and assembler macros," on page 215 for more details about how it is used. As mentioned above, "finish and reset" processing will occur as a part of this load operation. However, the reset will occur through a feature control operation, using the current feature set. In this way, the current feature flags for the parse instance are not altered by the OSR load control operation.
- The entities and references operation allows the caller to specify additional processing with regard to entity and character references. The GXLYCTL macro provides the interface for passing the information to the control function. As mentioned above, "finish and reset" processing will occur as part of this control

operation. However, the reset will occur through a feature control operation, using the current feature set. In this way, the current feature flags for the parse instance are not altered by the entities and references control operation.

- When document fragment parsing operation is enabled, the z/OS XML parser will no longer accept non-fragmented documents. If the caller wants to parse a complete document after enabling document fragment parsing, this service must be called again to disable document fragment parsing.
- When document fragment parsing is enabled, the well-formedness checking in the subsequent parsing will be confined to the scope of the document fragment. Well-formedness checking is performed again on the whole document when document fragment parsing is disabled.
- When document fragment parsing is enabled, a whitespace token will be placed into the output buffer when whitespace is parsed at the end of the input buffer for each document fragment.
- The OSR context is unaffected by document fragment parsing. Any OSR that is loaded when document fragment parsing is enabled will still be loaded for the parse of the fragment, and will remain loaded if fragment parsing is disabled.

For a list of properties and resources reset by the control functions, see "Properties and resources reset by control functions" on page 56.

GXL1CTL (GXL4CTL) features and functions

XEC_CTL_FIN

Description

This indicates that the caller wishes to end the current parse at the current position in the XML document. The PIMA is re-initialized to allow it to be used on a new parse request. To free up all resources associated with the parse instance, the caller should use the termination service. If the caller issues this control operation after document fragment parsing is enabled, then this control operation will disable document fragment parsing and re-initialize the PIMA for a new parse request. The loaded fragment context will remain in storage and become active when fragment mode is enabled.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

PIMA

Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA which has been previously initialized with a call to GXL1INI (GXL4INI)).

ctl_option
Supplied parameter

Type: Integer

Length:
Fullword

The name of a fullword that contains an integer value initialized to XEC_CTL_FRAGMENT_PARSE.

ctl_data
Supplied and returned parameter

Type: Address

Length:
Fullword (Doubleword)

This parameter must contain the address of a fullword (doubleword) where the service will store the address of the diagnostic area, which is mapped by macro GXLYXD. This provides additional information that can be used to debug problems in data passed to the z/OS XML parser. The diagnostic area resides within the PIMA, and will be overlaid on the next call to the z/OS XML parser.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the return code.

reason_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**

All parameters in the parameter list are required.

XEC_CTL_FEAT

Description

This indicates that the caller wishes to re-initialize the z/OS XML parser, as with the reset-and-finish function above, and in addition, that the caller wishes to reset some of the feature flags used during the parse.

Note: The following feature flags are not supported by this service:
- XEC_FEAT_JST_OWNS_STORAGE
- XEC_FEAT_RECOVERY
- XEC_FEAT_VALIDATE
- XEC_FEAT_SCHEMA_DISCOVERY
- GXLHXEC_FEAT_XDBX_INPUT

Make sure that these feature flags are turned to the OFF state before calling this service to set the feature flags. If these features need to be changed (for example, if switching between validating and non-validating parses), the parse instance must be terminated and re-initialized with the required feature settings.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

`PIMA`

Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

`ctl_option`

Supplied parameter

Type: Integer

Length:
Fullword

The name of a fullword containing an integer value initialized to XEC_CTL_FEAT.

`ctl_data`

Supplied and returned parameter

Type: Address

Type: Fullword (Doubleword)

This parameter must contain the address of a fullword (doubleword), which is mapped by macro GXLYXFT. See "gxlhxft.h (GXLYXFT) - mapping of the control feature input output area" on page 218 for more information on this macro.

The XFT_FEAT_FLAGS parameter is an input parameter to the API and contains the value of feature flags to be used in the subsequent parse. It is defined as follows:

XEC_FEAT_STRIP_COMMENTS
This effectively strips comments from the document by not returning any comments in the parsed data stream. Default: off.

XEC_FEAT_TOKENIZE_WHITESPACE
This sets the default token value for white space preceding markup in the root element to an explicit white space value. Default: off – white space is returned as character data.

XEC_FEAT_CDATA_AS_CHARDATA
This returns CDATA in records with a CHARDATA token type. The

content of these records may contain text that would normally have to be escaped to avoid being handled as markup. Default: off.

XEC_FEAT_SOURCE_OFFSETS
This feature is used to include records in the parsed data stream which contain offsets to the corresponding structures in the input document. Default: off.

XEC_FEAT_FULL_END
This feature is used to expand the end tags to include the local name, prefix and URI corresponding to the qname on the end tag. Default: off.

If none of the features are required, pass the name of a fullword field containing zero. Do not construct a parameter list with a zero pointer in it.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the return code.

reason_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

XEC_CTL_LOAD_OSR

Description

This indicates that the caller wants to load and use a given Optimized Schema Representation (OSR) during a validating parse. If the parse prior to invoking this operation returned a XRSN_NEED_OSR, this operation will not perform reset and finish processing.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

PIMA
Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_option
Supplied parameter

Type: Integer

Length:
Fullword

The name of fullword containing an integer value initialized to XEC_CTL_LOAD_OSR.

ctl_data
Supplied and returned parameter

Type: Address

Length:
Fullword (Doubleword)

This indicates that the caller wants to load and use a given Optimized Schema Representation (OSR) during a validating parse. Once an OSR has been loaded, it remains in use for all validating parse requests until a different OSR is provided by calling this service again.

This parameter must contain the address of an area containing information about the OSR to load. This area is mapped by GXLYXOSR. See "gxlhxosr.h (GXLYXOSR) - mapping of the OSR control area" on page 218 for more information about the macro.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the return code.

reason_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

XEC_CTL_QUERY_MIN_OUTBUF

Description

This indicates that the caller is requesting the control service to return the minimum output buffer size required for subsequent parse to complete without returning an XRSN_BUFFER_OUTBUF_SMALL reason code. This value is returned in the XD control block.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

PIMA

Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_option

Supplied parameter

Type: Integer

Length:
Fullword

The name of a fullword containing an integer value initialized to XEC_CTL_QUERY_MIN_OUTBUF.

ctl_data

Supplied and returned parameter

Type: Address

Length:
Fullword (Doubleword)

This parameter must contain the address of a fullword (doubleword) where the service will store the address of the diagnostic area, which is mapped by macro GXLYXD. The field XD_MIN_OB contains the minimum output buffer size required on the next parse. If some failure other than XRSN_BUFFER_OUTBUF_SMALL occurred prior to this call, XRSN_CTL_SEQUENCE_INCORRECT will be returned. The XD area will not be returned.

The diagnostic area resides within the PIMA, and will be overlaid on the next call to the z/OS XML parser.

return_code

Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the return code.

reason_code

Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

XEC_CTL_ENTS_AND_REFS

Description

This indicates that the caller is requesting additional flexibility when processing character or entity references. When this option is specified, the *ctl_data* parameter must also be utilized to specify the specific enhancement being requested. See the *ctl_data* section below for more information.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

PIMA
Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_option
Supplied parameter

Type: Integer

Length:
Fullword

The name of a fullword containing an integer value initialized to XEC_CTL_ENTS_AND_REFS.

ctl_data
Supplied and returned parameter

Type: Address

Length:
Fullword (Doubleword)

This parameter must contain the address of an area that contains information about what reference operations are to be processed. This area is mapped by the XEAR data structure in file GXLYCTL.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the return code.

reason_code
Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

XEC_CTL_LOAD_FRAG_CONTEXT

Description

This indicates that the caller wants to load fragment context into the z/OS XML parser. This service allows the caller to load namespace binding information and fragment paths for document fragment parsing. Namespace binding information is optional. Fragment path is required . This service must be issued prior to a XEC_CTL_FRAGMENT_PARSE control operation that enables document fragment parsing. If fragment context is already loaded from a prior XEC_CTL_LOAD_FRAG_CONTEXT control operation and this service is called again, the new fragment context will overlay the previously loaded context. This control operation will not cause finish/reset processing to take place.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

PIMA
Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_option
Supplied parameter

Type: Integer

Length:
Fullword

The name of a fullword containing an integer value initialized to XEC_CTL_LOAD_FRAG_CONTEXT.

ctl_data

Supplied and returned parameter

Type: Address

Length:
Fullword (Doubleword)

This parameter must contain a pointer to where the service will locate the address of the document fragment context structure, which is mapped by the macro GXL1CTL (GXL4CTL). The name of the data structure is GXLXFC. This structure allows the caller to provide the fragment path and namespace binding information to assist document fragment parsing.

To validate an element during document fragment parsing, the fragment path represents the path from the root element of the complete document to the root element of the fragment, which consists of prefixes and localnames. To validate an attribute during fragment parsing, the fragment path represents the path from the root element of the complete document to the desired attribute name. The fragment path is required in order to perform validation in fragment parsing.

The fragment path syntax is defined below:

```
FragmentPath ::= ('/' ElementName)* FragmentData
FragmentData ::= '/' ElementName ('/@' AttributeName)?
ElementName ::= QName
AttributeName ::= QName
```

Namespaces bindings allow unique strings of text that identify a given space of names to be represented by a prefix. This allows references to elements with the same name to be differentiated, based on the namespace to which they belong. These bindings may not be present in the document fragment, and often these bindings exist in the ancestor elements' start tag that is not part of the document fragment. The caller can provide a complete context containing multiple namespace bindings in the GXLXFC structure. The namespace binding is optional information.

However, if there is an XML instance document that uses a default namespace, the caller must still specify a prefix on the element names in the fragment path. The caller must also specify this prefix along with the namespace URI in the namespace binding information. The actual prefix does not matter; only the namespace URI matters, but the prefix will associate each element in the fragment path with the correct namespace.

Note:

1. All the strings for fragment path and namespace binding passed into the XEC_CTL_LOAD_FRAG_CONTEXT control call needs to be in the encoding of the z/OS XML parser configured at initialization time.
2. If the caller disables document fragment parsing, the namespace contexts loaded through the XEC_CTL_LOAD_FRAG_CONTEXT control call will be removed and will not be available during the non-fragment parsing mode.
3. When the caller issues a XEC_CTL_LOAD_FRAG_CONTEXT control call to load namespace contexts, the namespace contexts will be available when the z/OS XML parser switches into fragment parsing mode. The namespace contexts will only get unloaded and replaced if the caller terminates the parser or issues XEC_CTL_LOAD_FRAG_CONTEXT control call again to load new namespace contexts.

`return_code`
Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the return code.

`reason_code`
Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

XEC_CTL_FRAGMENT_PARSE

Description

This indicates that the caller wants to either enable or disable document fragment parsing. This service will decide whether to enable or disable document fragment parsing based on the XFP_FLAGS_FRAGMENT_MODE bit set in the *ctl_data* parameter. Document fragment parsing is disabled by default. This control operation will not cause finish/reset processing to take place. If the caller wants to parse a new complete XML document, a XEC_CTL_FIN control operation must be called prior to a new parse request. If any error with return code greater than 4 has occurred during document fragment parsing, a XEC_CTL_FIN control operation must be issued in order to resume parsing. Calling the XEC_CTL_FIN control operation will disable the document fragment parsing and unload all fragment contexts.

Note:

1. Document fragment parsing can only be enabled once before disabling. Likewise, document fragment parsing can only be disabled once before enabling.
2. If the caller disables document fragment parsing, the parse will end and the caller is allowed to parse a new document.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

`PIMA`
Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_option
Supplied parameter

Type: Integer

Length:
Fullword

The name of a fullword containing an integer value initialized to XEC_CTL_FRAGMENT_PARSE.

ctl_data
Supplied and returned parameter

Type: Address

Length:
Fullword (Doubleword)

This parameter must contain a pointer to where the service will locate the address of the document fragment parsing structure, which is mapped by the macro GXLCTL. The name of the data structure is GXLXFP. This structure allows the caller to specify whether to enable or disable document fragment parsing through the XFP_FLAGS_FRAGMENT_MODE bit set in the XFP_FLAGS field. Document fragment parsing is disabled by default.

The XFP_XD_PTR is where the service will store the address of the diagnostic area, which is mapped by macro GXLYXD. This provides additional information that can be used to debug problems in data passed to the z/OS XML parser. The diagnostic area resides within the PIMA, and will be overlaid on the next call to the z/OS XML parser.

Tips:

- To enable document fragment parsing, set the XFP_FLAGS_FRAGMENT_MODE bit to on.
- To disable document fragment parsing, set the XFP_FLAGS_FRAGMENT_MODE bit to off.

Note:

1. When the caller validates an attribute during fragment parsing, the document fragment passed to the parser should contain only the desired attribute's value.
2. When the caller re-enables document fragment parsing after it has been disabled, and without calling load fragment context again, the previous loaded fragment context will be utilized in this new fragment parse. This includes the fragment path and any namespace binding information.
3. The OSR must be loaded by way of the XEC_CTL_LOD_OSR control call prior to enabling fragment parsing.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the return code.

reason_code

Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

XEC_CTL_RESTRICT_ROOT

Description

This operation indicates that the caller wishes to restrict the root element name on the next parse. If the root element name is not any of those listed in the GXLXRR data area, this call will cause the parse to stop. This operation will reset the PIMA.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

PIMA

Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_option

Supplied parameter

Type: Integer

Length:
Fullword

The name of the fullword containing an integer value initialized to XEC_CTL_RESTRICT_ROOT.

ctl_data

Supplied and returned parameter

Type: Address

Length:
Fullword

This parameter contains the address of an area with information about the restricted root element. This area is mapped by macro GXLXRR. This provides a list of names that must contain the name of the root element in order for the validating parse to succeed.

`return_code`

Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the return code.

`reason_code`

Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

XEC_CTL_ERROR_HANDLING

Description

With this control operation, the caller can do the following for a validating parse:

- Enable the creation of auxiliary records which can include the location of an error in the XML document, the string which is in error, and also a possible expected string.
- Enable position indexes to be present in the error location path in order to facilitate locating the error.

For a non-validating parse, it can be used to:

- Enable the ability to continue parsing when an undefined prefix is encountered on an element or attribute. The “prefix:local name” will be treated as the local name.
- Request an auxiliary information record that contains the tolerated return and reason codes and the error offset.

Syntax

```
call gxl1ctl,(PIMA,
              ctl_option,
              ctl_data,
              return_code,
              reason_code);
```

Parameters

`PIMA`

Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA) which has been previously initialized with a call to the initialization service.

ctl_option
Supplied parameter

Type: Integer

Length:
Fullword

The name of the fullword containing an integer value initialized to XEC_CTL_ERROR_HANDLING.

ctl_data
Supplied and returned parameter

Type: Address

Length:
Fullword

This parameter contains the address of an area with information about the error string. This is the XERR data structure mapped by macro GXLXRR.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the return code.

reason_code
Returned parameter

Type: Integer

Length:
Fullword

The name of the fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

The enhanced error information for a validating parse is returned by way of the XERR_XD_PTR and is where the service will store the address of the diagnostic area, which is in the macro GXLXRR. The XD_LastOutput field is a pointer to the data area containing these records. This data area is within the PIMA and is formatted in the same manner as a normal output buffer.

The XEC_TOLERATED_ERROR auxiliary info record for a non-validating parse is returned in the output buffer. In the event that source offset auxiliary records are also being returned, this record will immediately follow those records for the element or attribute in the output buffer.

In addition to enabling or disabling the enhanced error features, this control option will perform a reset function. The following properties and resources will be reset by this control option:

- Fragment mode (validating parse only)
- Start of the XML document
- Error state

GXL1INI (GXL4INI) — initialize a parse instance

Description

The GXL1INI (GXL4INI) callable service initializes the PIMA and records the addresses of the caller's system service routines (if any). The PIMA storage is divided into the areas that will be used by the z/OS XML parser to process the input buffer and produce the parsed data stream.

Performance Implications

The initialization of structures used by the z/OS XML parser in the PIMA is only done once per parse and is therefore unlikely to affect performance. The caller may choose to reuse the PIMA after each parse to eliminate the overhead of storage allocation and the page faults that occur when referencing new storage. In this case, a control operation is required to reset the necessary fields in the PIMA before parsing can continue.

Syntax

```
call gxl1ini,(PIMA,
             PIMA_len,
             ccsid,
             feature_flags,
             sys_svc_vector,
             sys_svc_parm,
             return_code,
             reason_code)
```

Parameters

PIMA

Supplied parameter

Type: Character string

Length:
determined by the PIMA_len parameter

The name of the Parse Instance Memory Area (PIMA). The PIMA must be aligned on a doubleword boundary, otherwise, results are unpredictable. See the “Usage notes” on page 134 below for additional details on the use of this area.

PIMA_len

Supplied parameter

Type: Integer

Length:
Fullword (Doubleword)

The name of an area containing the length of the Parse Instance Memory Area. This service validates the length of this area against a minimum length value. The minimum length of the PIMA depends on whether or not validation will be performed during the parse. This minimum length value can be found in:

- XEC_NVPARSE_MIN_PIMA_SIZE (non-validating parse)
- XEC_VPARSE_MIN_PIMA_SIZE (validating parse)

ccsid

Supplied parameter

Type: Integer

Length:
Fullword

The Coded Character Set IDentifier (CCSID) that identifies the document's character set. The CCSID value in this parameter will override any character set or encoding information contained in the XML declaration of the document. A set of CCSID constants for supported encodings has been declared in GXLYXEC. See Appendix I, "Supported encodings," on page 239 for a full list of supported encodings.

feature_flags

Supplied parameter

Type: Integer

Length:
Fullword

The name of a fullword that contains an integer value representing one or more of the following z/OS XML parser features. OR these flags together as needed to enable features. Choose any of the following:

- **XEC_FEAT_STRIP_COMMENTS** - effectively strip comments from the document by not returning any comments in the parsed data stream.
- **XEC_FEAT_TOKENIZE_WHITESPACE** - set the default token value for white space preceeding markup within the context of the root element to an explicit white space value. Use this value in conjunction with the special xml:space attribute to determine how such white space gets classified.
- **XEC_FEAT_CDATA_AS_CHARDATA** - return CDATA in records with a CHARDATA token type. The content of these records may contain text that would normally have to be escaped to avoid being handled as markup.
- **XEC_FEAT_JST_OWNS_STORAGE** - allocate storage as Job Step Task (JST) related instead of task related. See the "Usage notes" on page 134 below for more information.
- **XEC_FEAT_RECOVERY** - this turns on the recovery routine.

  Note: The following only applies when the feature flag is ON:

  - If running in SRB mode, an error message will be returned to the caller.
  - If a parse request is made in SRB mode, the parse will fail.
  - If there is an FRR, an error message will be returned to the caller during the parse step.
- **XEC_FEAT_SOURCE_OFFSETS** - this includes records in the parsed data stream which contain offsets to the corresponding structures in the input document.
- **XEC_FEAT_FULL_END** - this expands the end tags to include the local name, prefix and URI corresponding to the qname on the end tag.

- **XEC_FEAT_VALIDATE** - this initializes a parse instance that allows for validation during parsing. See the usage notes below for details on validation.
- **XEC_FEAT_SCHEMA_DISCOVERY –** report schema location information and allow for an OSR to be loaded once the information has been reported. XEC_FEAT_VALIDATE must also be enabled, otherwise GXL1INI (GXL4INI) will return an error. See "Usage notes" on page 137 for more information on schema discovery. Default: off
- **XEC_FEAT_XDBX_INPUT** - indicates that the data presented to z/OS XML in the input buffer is in XDBX binary XML form, rather than conventional text. This feature requires that XEC_FEAT_VALIDATE is also set, and that the encoding specified in the CCSID parameter is UTF-8. See "Usage notes" on page 134 for more information on XDBX input streams. Default: off.

Note: By using the values of off (zero), W3C XML compliant output is generated. Turning on options XEC_FEAT_STRIP_COMMENTS, XEC_FEAT_TOKENIZE_WHITESPACE and XEC_FEAT_CDATA_AS_CHARDATA will cause the output to vary from standard compliance.
If none of the features are required, pass the name of a fullword field containing zero. Do not construct a parameter list with a zero pointer in it.

sys_svc_vector

Supplied parameter

Type: Structure

Length:
Variable

The name of a structure containing a count of entries that follow and then a list of 31 (64) bit pointers to system service routines. Specify the name of a word containing 0 if no services are provided. See the Chapter 8, "z/OS XML System Services exit interface," on page 145 chapter for more details.

sys_svc_parm

Supplied parameter

Type: Address

Length:
Fullword (Doubleword)

The name of a parameter which is passed to all system service exits. This provides for communication between the z/OS XML parser caller and its exit routines. Specify the name of a location containing 0 if no parameter is required for communication.

return_code

Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the return code.

reason_code

Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in macro GXLYXR. For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

For an AMODE 31 example using this callable service, see “GXL1INI example” on page 222. For an AMODE 64 example using this callable service, see “GXL4INI example” on page 226.

Usage notes

- The z/OS XML parser creates a variety of control blocks, tables, stacks, and other structures in the Parse Instance Memory Area. The caller must provide an area that is at least as large as constant **XEC_MIN_PIMA_SIZE**. In the event that this area is not large enough to parse the input document, the z/OS XML parser will allocate additional memory using either the default memory allocation mechanism or the memory allocation exit that the caller has provided.
- When the PIMA is reused for subsequent parses, the same features, ccsid and service exits will apply. If any of these values need to change, you should terminate the parse instance (call GXL1TRM (GXL4TRM)) and call GXL1INI (GXL4INI) again with the options you require.
- When the **XEC_FEAT_TOKENIZE_WHITESPACE** feature is set, the default classification for white space that precedes markup within the context of the root element will be **XEC_TOK_WHITESPACE**. This token type is returned if either the white space being parsed does not have an xml:space context, or if the xml:space setting is 'default'. When the tokenize white space feature is not enabled, or if the white space does not precede markup, this white space will be returned in the parsed data stream containing character data with a token type of **XEC_TOK_CHAR_DATA**.
- The **XEC_FEAT_JST_OWNS_STORAGE** feature only applies to callers running in non-cross memory task mode who take the option of allowing the z/OS XML parser to allocate additional storage as needed. This feature should be specified when PIMAs are used on multiple tasks in order to prevent task termination from causing storage extents to be freed before the z/OS XML parser is done using them.
- Before requesting the initialization of a validating parse instance, the validation function must be loaded – either through one of the methods that the system provides, or by the z/OS XML load service. Failure to do so will result in an error indicating that the function is not available. See the description of “GXL1LOD (GXL4LOD) — load a z/OS XML function” on page 142 for more information.

- Be sure that the size of the PIMA provided is large enough for the XML processing function, either validating or non-validating parse, that will be performed. Also, make sure that there is an appropriate minimum PIMA size constant defined for each in GXLYXEC.
- The performance of a validating parse will be best when the parsed document is in the UTF-8 encoding. The other encodings supported by z/OS XML System Services are also supported during a validating parse, but there is significant additional overhead that will impact performance.
- For usage notes on parsing XDBX input streams, see "Parsing XDBX input streams" on page 47.

GXL1PRS (GXL4PRS) — parse a buffer of XML text

Description

The GXL1PRS callable service parses a buffer of XML text and places the result in an output buffer.

Performance Implications

Ideal performance will be obtained when the PIMA is sufficiently large to contain all the needed data structures, and the input and output buffers are large enough to process the entire XML document. During the parsing process, the z/OS XML parser constructs persistent information in the PIMA that can be reused within a parse instance. If the caller is going to process multiple documents that contain similar sets of symbols (namespaces and local element and attribute names in particular), then reusing the PIMA will improve performance during the processing of subsequent documents. If this behavior is not required, the PIMA should be cleaned up by calling GXL1TRM (GXL4TRM) and reinitialized by calling GXL1INI (GXL4INI) before using the PIMA for another parse request.

Syntax

```
call gxl1prs,(PIMA,
             option_flags,
             input_buffer_addr,
             input_buffer_bytes_left,
             output_buffer_addr,
             output_buffer_bytes_left,
             return_code,
             reason_code)
```

Parameters

PIMA

Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA which has been previously initialized with a call to GXL1INI (GXL4INI)).

option_flags

Supplied parameter

Type: Integer

Length:
Fullword

Specify a word of zeroes for this parameter. In the future, this field will allow options to be compatibly added to the service.

input_buffer_addr
Supplied and returned parameter

Type: Address

Length:
Fullword (Doubleword)

The name of a fullword (doubleword) that contains the address of the buffer with the XML text to parse. The z/OS XML parser updates this parameter to provide important return information when control returns to the caller. See the “Usage notes” on page 137 below for details.

input_buffer_bytes_left
Supplied and returned parameter

Type: Integer

Length:
Fullword (Doubleword)

The name of a fullword (doubleword) that contains the number of bytes in the input buffer that have not yet been processed. The z/OS XML parser updates this parameter to provide important return information when control returns to the caller. See the “Usage notes” on page 137 for details.

output_buffer_addr
Supplied and returned parameter

Type: Address

Length:
Fullword (Doubleword)

The name of a fullword (doubleword) that contains the address of the buffer where the z/OS XML parser should place the parsed data stream. The z/OS XML parser updates this parameter to provide important return information when control returns to the caller. See the “Usage notes” on page 137 for details.

output_buffer_bytes_left
Supplied and returned parameter

Type: Integer

Length:
Fullword (Doubleword)

The name of a fullword (doubleword) that contains the number of available bytes in the output buffer. When the z/OS XML parser returns control to the caller, this parameter will be updated to indicate the number of unused bytes in the output buffer. This buffer must always contain at least a minimum number of bytes as defined by the **XEC_MIN_OUTBUF_SIZE** constant, declared in macro GXLYXEC. This service will validate the length of this area against this minimum length value.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the return code.

`reason_code`
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in macro GXLYXR. For reason code descriptions, also see Appendix B, "Reason codes listed by value," on page 161.

Example

For an AMODE 31 example using this callable service, see "GXL1PRS example" on page 223. For an AMODE 64 example using this callable service, see "GXL4PRS example" on page 227.

Usage notes

- When the z/OS XML parser returns successfully to the caller, the input and output buffer addresses will be updated to point to the byte after the last byte successfully processed. The *input_buffer_bytes_left* and *output_buffer_bytes_left* parameters will also be updated to indicate the number of bytes remaining in their respective buffers. In the event of an error caused by a problem with the document being parsed, the input buffer address will point to the byte of the input stream where the problem was detected, and the associated bytesleft value will indicate the same position in the buffer. An error record will be written to the parsed data stream indicating the nature of the problem, and the output buffer address and bytesleft fields will point to the next available byte, as in the success case. See Chapter 4, "Parsing XML documents," on page 11 for more information about how input and output buffers are managed between the caller and z/OS XML parser.
- In cases where parsing terminates because of an error, the z/OS XML parser will often have partially processed an item from the input document before returning to the caller. The caller has the option of retrieving the address of the diagnostic area using the GXL1CTL (GXL4CTL) service. The **XD_LastRC/XD_LastRsn** return/reason code combination will contain an indication of the item being parsed. Retrieving the reason code in this manner is an example of the indirect method for obtaining a specific reason code.
- The z/OS XML parser will always check that the output buffer length passed to it is greater than the required minimum (**XEC_MIN_OUTBUF_SIZE**). If this minimum length requirement is not met, the z/OS XML parser will return with

a return/reason code of **XRC_FAILURE/XRSN_BUFFER_OUTBUF_SMALL**. Output buffer spanning will only occur if the caller meets the minimum output buffer length requirement when the z/OS XML parser is invoked. Once parsing begins, and the buffer info record has been written to the output buffer, buffer spanning is enabled. The caller will then receive an end-of-output-buffer indication when the end of the output buffer is reached. In addition, many non-splittable records will be larger than the minimum output buffer size. If there is not enough space in the output buffer for the first record, then XRC_FAILURE/XRSN_BUFFER_OUTBUF_SMALL will be returned. Therefore, it's recommended that the output buffer sizes should be large enough to fit the largest record that is expected to be encountered.

- When schema discovery is enabled, XRSN_NEED_OSR may be returned from a parse request. This signifies that the parser has finished parsing the root element start tag and has returned enough information to identify a schema. At this point, a load OSR operation may be performed without the operation resetting the parser. If a reset is intended, then an explicit call to GXL1CTL (GXL4CTL) with the XEC_CTL_FIN option must be made prior to the next parse.
- When the z/OS XML parser returns a failure to the caller, GXL1CTL (GXL4CTL) must issue the control option XEC_CTL_FIN in order to continue document fragment parsing or non-fragment parsing.
- When the z/OS XML parser returns successfully to the caller, it indicates the end of the provided input buffer was reached and the parsed XML data is well-formed. When document fragment parsing is enabled, this service will confine well-formedness checking to the scope of the document fragment. This behavior is enabled and disabled through use of the XEC_CTL_FRAGMENT_PARSE control operation.
- If the caller disables fragment parsing by calling GXL1CTL (GXL4CTL) with the control option XEC_CTL_FRAGMENT_PARSE, and the z/OS XML parser returns to the input or output buffer during document fragment parsing, an error will occur.
- If the caller performs validation in fragment parsing, the input buffer must be restricted to a single element and its descendants, optionally followed by comments and processing instructions.
- If the caller performs document fragment parsing on an attribute, the input buffer should only contain the desired attribute’s value. See the following example:

```
XML Document: <root> <pfx:ln attr="attributeValue"/> </root>
Fragment Path = /root/pfx:ln/@attr
Input Buffer = attributeValue
```

GXL1QXD (GXL4QXD) — query an XML document

Description

This service allows a caller to obtain the XML characteristics of a document. The XML characteristics are either the default values, the values contained in an XML declaration or a combination of both.

Performance Implications

There are no performance implications.

Syntax

```
call gxl1qxd,(work_area,
             work_area_length,
             input_buffer,
             input_buffer_length,
             return_data,
             return_code,
             reason_code)
```

Parameters

work_area

Supplied parameter

Type: Character string

Length:
Variable

The name of a work area. The work area must be aligned on a doubleword boundary. If not on a doubleword boundary, results are unpredictable. See the "Usage notes" on page 140 below for additional details on the use of this area.

work_area_len

Supplied parameter

Type: Integer

Length:
Fullword (Doubleword)

The name of an area containing the length of the work area. The minimum length of this area is declared as a constant **XEC_MIN_QXDWORK_SIZE** in macro GXLYXEC. This service validates the length of this area against this minimum length value.

input_buffer

Supplied parameter

Type: Character string

Length:
Variable

The name of an input buffer containing the beginning of the XML document to process. See the "Usage notes" on page 140 below for details.

input_buffer_length

Supplied parameter

Type: Integer

Length:
Fullword (Doubleword)

The name of an area containing the length of the input buffer.

return_data

Returned parameter

Type: Address

Length:
Fullword (Doubleword)

The name of a fullword (doubleword) where the service will return the address of the data which describes the XML document characteristics. This return information will contain values that are either extracted from the XML declaration or defaulted according to the XML standard. This return area is mapped by macro GXLYQXD (see “gxlhqxd.h (GXLYQXD) - mapping of the output from the query XML declaration service” on page 216), and is located within the work area specified by the work_area parameter. The caller must not free the work_area until it is done referencing the data returned from this service.

return_code

Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the return code.

reason_code

Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in macro GXLYXR (see “gxlhxr.h (GXLYXR) - defines the return codes and reason codes” on page 218). For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

Usage notes

- The input buffer passed to this service must contain the beginning of the XML document to process. This service will look for any XML declaration that is present and extract the version, encoding, and standalone value that are present. In the event that the document does not contain an XML declaration, or a given value is missing from the declaration, this service will return an appropriate default, as specified by the XML standard. On success, the return data address for this service will contain a pointer into the work area where the return data has been collected.
- Unlike the GXL1PRS (GXL4PRS) or GXL1CTL (GXL4CTL) services that must be performed within a parse instance, this service does not require any of the internal resources that the z/OS XML parser creates in the PIMA during initialization. It does not advance the input pointer or modify the state of the

parse in any way. It is a simple standalone service that allows a caller to query important information about the document before establishing a parse instance and performing the parse.

- Buffer spanning is not supported by this service, as it is by GXL1PRS (GXL4PRS). If either the input buffer or the work area are too small, this service will terminate with an appropriate return/reason code.
- This service is useful for checking to see if a conversion to one of the supported encodings is required before parsing the document.
- Encoding names supported include the IANA recommended names which have corresponding IBM CCSID values.
- This service does not provide full well-formedness checking of the input it processes.

GXL1TRM (GXL4TRM) — terminate a parse instance

Description

The GXL1TRM callable service releases all resources obtained (including storage) by the z/OS XML parser and resets the PIMA so that it can be re-initialized or freed.

Performance Implications

There are no performance implications.

Syntax

```
call gxl1trm,(PIMA,
             return_code,
             reason_code)
```

Parameters

PIMA

Supplied parameter

Type: Character string

Length:
Variable

The name of the Parse Instance Memory Area (PIMA which has been previously initialized with a call to GXL1INI (GXL4INI)).

return_code

Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the return code.

reason_code

Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both also set as output parameters. The value of the reason code is undefined when the return code has no associated reasons. Return and reason codes are defined in macro GXLYXR. For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Example

For an AMODE 31 example using this callable service, see “GXL1TRM example” on page 223. For an AMODE 64 example using this callable service, see “GXL4TRM example” on page 228.

Usage notes

Termination can be requested any time the caller gets control back from the z/OS XML parser. This service does not free the Parse Instance Memory Area (PIMA) as a part of termination. If the caller's recovery gets control while a parse is still in progress, the caller should invoke this termination service to clean up resources.

GXL1LOD (GXL4LOD) — load a z/OS XML function

Description

Load a module that implements a z/OS XML function into storage.

Performance Implications

None.

Syntax

```
call gxl1lod(function_code,
             function_data,
             return_code,
             reason_code)
```

Parameters

`function_code`
Supplied parameter

Type: Integer

Length:
Fullword

This parameter identifies the z/OS XML function to load. It is the name of a fullword that contains an integer value representing one of the following functions:

XEC_LOD_VPARSE
The validating parse function.

See the GXLYXEC macro for the list of function code constants.

`function_data`
Supplied parameter

Type: Address

Length:
Fullword (doubleword)

Specify a word of zeroes for this parameter.

`return_code`
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the return code.

`reason_code`
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the service stores the reason code. The reason code is only relevant if the return code is not **XRC_SUCCESS**.

All parameters in the parameter list are required.

Return and Reason Codes:

On return from a call to this service, register 15 will contain the return code. The return and reason code are both set as output parameters. The value of the reason code is undefined when the return code is 0 (XRC_SUCCESS). Return and reason codes are defined in macro GXLYXR, and are dependent on the control function specified by the caller. For reason code descriptions, also see Appendix B, “Reason codes listed by value,” on page 161.

Usage notes

This load step is not required when performing a non-validating parse. This operation is only required when using the validating parser. The caller does have the option of loading the load module for the specified function without using this service - either through the z/OS LOAD macro, or by putting it in LPA or the extended LPA. Both the LOAD macro and calls to this service are not allowed when running in an SRB. The use of either interface must be performed in the task before entering SRB mode.

If the required z/OS XML function is made available, either by LOADing the executable load module for it or putting the load module in LPA, this service is not required. Documentation on the LOAD macro can be found in z/OS MVS

Programming: Assembler Services Reference, Volume 2, and information on how to load modules into LPA can be found in z/OS MVS Initialization and Tuning Guide.

The load modules associated with each function are as follows:

Table 31. Load modules

Function code	Function performed	Load module name
XEC_LOD_VPARSE	validating parser function	GXLIMODV

There is no unload service to perform the converse of this function, and none of the other z/OS XML System Services cause the z/OS XML parser to be unloaded. The z/OS XML parser load module will remain in the caller's address space even if the parser is terminated or reset. If multiple parse requests are to be performed in the same address space, make sure to load the z/OS XML parser only once, regardless of whether those parse requests are performed using the same parse instance (PIMA) or not.

Chapter 8. z/OS XML System Services exit interface

The system services exit interface defines a series of exits that give the original caller of the GXL1PRS (GXL4PRS) service control over the way the z/OS XML parser acquires/releases resources, and over certain parser operations. The interface is implemented as a vector of addresses to routines that perform these operations. The first word in the vector is a count of the number of addresses which follow — both NULL addresses indicating that a specific exit is not present, and non–NULL addresses. If this count is zero, then the z/OS XML parser will use default services. Similarly, an entry in the system service vector may be left NULL, and the default service that corresponds to that entry will be used. For the storage allocation and deallocation exits, either both or neither exit must be specified. The addresses of the routines are 4 bytes when in AMODE 31 and 8 bytes when in AMODE 64. The mapping macro GXLYXSV (see "gxlhxsv.h (GXLYXSV) - mapping of the system service vector" on page 218) is available to help set up this structure.

Exit functions

The system services exit interface contains exits to perform the following functions:

- Allocate memory
- Free memory
- String identifier service — this is used to create a unique 4 byte numerical value (StringID) that corresponds to a string parsed from the document. This exit allows the caller to control the individual StringID values that the z/OS XML parser uses and serves as an efficient mechanism to communicate these values between caller and parser. If no StringID service is specified, StringIDs are not exploited by the z/OS XML parser and the parsed data stream will contain only length/value pairs for all parsed strings.

These exits are all passed the address of a system service work area. This work area is storage that was obtained by the caller and can be used to store any information which may make communication between the caller and the exits easier.

Common register conventions

The following are common register conventions for all of the system service interface exits:

Input registers

When the z/OS XML parser invokes an exit, these registers have the following meaning:

Table 32. System services input register conventions

Register	Contents
1	Address of a standard parameter list containing 31 (64) bit addresses.
13	Address of a 72 (144) byte save area.
14	Return address

Table 33. System services input access register conventions

Access Register	Contents
0-15	Unpredictable

Output registers

When an exit returns to the z/OS XML parser, these registers have the following meaning:

Table 34. System services output register conventions

Register	Contents
0-1	Unpredictable
2-13	Unpredictable
14	Return address
15	Unpredictable

Table 35. System services output access register conventions

Register	Contents
0-15	Unpredictable

The z/OS XML parser saves all general purpose and access registers prior to calling the user exit. The user exit must simply return to the address in register 14. The save area provided can be used for any needs of the exit.

Environmental requirements

The system services exit interface exits are called in the same environment in which the z/OS XML parser was invoked. This means the following:

Minimum authorization
any state and any PSW key

Dispatchable unit mode
Task or SRB

Cross memory mode
Any PASN, any HASN, any SASN

AMODE
31-bit (64-bit)

ASC mode
primary

Interrupt status
enabled for I/O and external interrupts

Locks no locks held

Control parameters
Control parameters and all data areas the parameter list points to are addressable from the current primary address space.

Restrictions

These exit routines must not call any of the services provided in the z/OS XML parser API, either directly or indirectly.

These exit routines are required to use linkage OS. As a result, they will need to be written in assembler and not C or C++.

The two storage exits, "GXLGST31 (GXLGST64) — get memory" and "GXLFST31 (GXLFST64) — free memory" on page 149, must be called together. They cannot be called independently of one another.

Although the actual name of the entry points to each of these exit services may be anything the caller wishes, the z/OS XML parser will call these services as if they had the interfaces listed below.

GXLGST31 (GXLGST64) — get memory

Description

This service allocates an area of memory of the size requested by the z/OS XML parser. The z/OS XML parser requests memory in large quantities and manages sub-allocations of this memory within the parser.

Performance Implications

There are no performance implications.

Syntax

```
call gxlgst31,(sys_svc_parm,
              memory_addr,
              memory_len,
              exit_diag_code,
              return_code,
              reason_code)
```

Parameters

sys_svc_parm

Supplied parameter

Type: Address

Length:
Fullword (Doubleword)

The address of the system service parameter (or zero) that was passed to the z/OS XML parser at initialization time.

memory_addr

Returned parameter

Type: Address

Length:
Fullword (Doubleword)

The address of a fullword (doubleword) where the memory allocation exit should store the address of the allocated memory. If the caller wants to terminate the parse, then it should set a nonzero return code.

memory_len

Supplied and Returned parameter

Type: Integer

Length:
Fullword (Doubleword)

A fullword that contains the length of the memory area requested by the z/OS XML parser. The exit is allowed to return an area of greater size and set this parameter to the length returned.

exit_diag_code
Returned parameter

Type: Integer

Length:
Fullword (Fullword)

The name of a fullword where the exit stores any diagnostic information (usually a reason code). This is stored in the diagnostic area and made available on the GXL1CTL (GXL4CTL) call.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the exit service stores the return code.

reason_code
Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the exit service stores the reason code.

Return and Reason Codes:

The z/OS XML parser uses the convention that the exit will provide a return code value of zero when successful. Any nonzero value indicates failure. If a nonzero return code is provided by the exit, the z/OS XML parser does not look at the reason code. Instead, the z/OS XML parser saves the reason code, along with the return code and the diagnostic code, in the extended diagnostic area so that the caller of the z/OS XML parser has access to it by calling GXL1CTL (GXL4CTL). The z/OS XML parser will provide return and reason codes to the caller in the event of a failure by the exit, or if the parser detects a problem with the storage returned from the exit.

For reason code descriptions, see Appendix B, “Reason codes listed by value,” on page 161.

Example

For an example using this exit service, see “GXLEGTM (GXLGST example)” on page 230. These examples are located in SYS1.SAMPLIB .

Default Implementation

If the exit is not provided, then the subpool used will be as follows:

- If running in SRB or cross memory mode, subpool 129 will be used. This is JST related and cannot be freed by unauthorized callers. The key will be the same as the key at the time the z/OS XML parser is invoked.
- If running in task mode (PSATOLD not zero), with PRIMARY=SECONDARY=HOME, then the subpool chosen will depend on the authorization state of the caller and on the specification of the XEC_FEAT_JST_OWNS_STORAGE feature on the GXL1INI (GXL4INI) call. If the caller is running in key 0-7 or supervisor state, they will be considered authorized.
  - Authorized and JST requested — subpool 129
  - Authorized and JST not requested — subpool 229
  - Unauthorized and JST requested — subpool 131
  - Unauthorized and JST not requested — subpool 0

  Note: If running on a subtask which is sharing subpool 0, then this storage will be owned by the task that owns subpool 0.

These choices of subpool will eliminate the possibility of the z/OS XML parser running in an authorized state while using problem key storage which could be freed and reallocated.

The CONTROL setting will be AUTH for authorized callers. This prevents the storage from being unallocated by an unauthorized caller in the same address space. The storage will be allocated in the caller's key.

GXLFST31 (GXLFST64) — free memory

Description

This service frees an area of memory previously obtained by the GXLGST31 (GXLGST64) service.

Performance Implications

There are no performance implications.

Syntax

```
call gxlfst31,(sys_svc_parm,
              memory_addr,
              memory_len,
              exit_diag_code,
              return_code
              reason_code)
```

Parameters

sys_svc_parm
Supplied parameter

Type: Address

Length:
Fullword (Doubleword)

The address of the system service parameter that was passed to the z/OS XML parser at initialization time.

memory_addr

Supplied parameter

Type: Address

Length:

Fullword (Doubleword)

The address of a fullword (doubleword) that contains the address of the memory to be freed.

memory_len

Supplied parameter

Type: Integer

Length:

Fullword (Doubleword)

A fullword (doubleword) that contains the length of the memory to be freed. Memory will always be freed in the same quantities under which it was allocated.

exit_diag_code

Returned parameter

Type: Integer

Length:

Fullword

The name of a fullword where the exit can store any diagnostic information (usually a reason code).

return_code

Returned parameter

Type: Integer

Length:

Fullword

The name of a fullword where the exit service stores the return code.

reason_code

Returned parameter

Type: Integer

Length:

Fullword

The name of a fullword where the exit service stores the reason code.

Return and Reason Codes:

The z/OS XML parser uses the convention that the exit will provide a return code value of zero when successful. Any nonzero value indicates failure.

For reason code descriptions, see Appendix B, “Reason codes listed by value,” on page 161.

Example

For an example using this exit service, see “GXLEFRM (GXLFST example)” on page 229. These examples are located in SYS1.SAMPLIB .

Default Implementation

The z/OS XML parser will free all memory obtained. Memory is freed in the same quantities under which it was allocated. See the MVS assembler services reference (SA22-7606) for more details on the STORAGE macro.

GXLSYM31 (GXLSYM64) — StringID service

Description

This service accepts an input string and performs a lookup for its corresponding symbol, which is identical to the string itself. If the symbol has been located, the exit returns the StringID associated with the symbol. If the string does not have a defined symbol, a symbol is created for the string and a StringID is assigned to it. The StringID is then returned to the z/OS XML parser.

Performance Implications

There are no performance implications.

Syntax

```
call gxlsym31,(sys_svc_parm,
              string,
              string_len,
              string_id,
              ccsid,
              exit_diag_code,
              return_code)
```

Parameters

sys_svc_parm

Supplied parameter

Type: Address

Length:
Fullword (Doubleword)

The address of the system service parameter that was passed to the z/OS XML parser at initialization time.

string

Supplied parameter

Type: Character string

Length:
determined by the string_len parameter

The string to return an ID for. The length of the string is variable, and is specified by the string_len parameter.

string_len

Supplied parameter

Type: Integer

Length:
Fullword

A fullword that contains the length of the string pointed to by the string parameter.

string_id

Returned parameter

Type: Unsigned integer

Length:
Fullword

The numeric identifier for the string. The range of valid values is 1 to 2 GB - 1. The value zero is reserved for use by the z/OS XML parser.

ccsid

Supplied parameter

Type: Integer

Length:
Fullword

The Coded Character Set IDentifier (CCSID) that identifies the character set of the string. The z/OS XML parser will provide the same CCSID in this parameter that the caller of the parser specified at parser initialization time.

exit_diag_code

Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword where the exit can store any diagnostic information (usually a reason code). This will be stored in the diagnostic area and made available on the GXL1CTL (GXL4CTL) call.

return_code

Returned parameter

Type: Integer

Length:
Fullword

The name of a fullword containing the return code. A return code value of zero means success; any nonzero return code indicates failure.

Return Codes:

The z/OS XML parser uses the convention that the exit will provide a return code value of zero when successful. Any nonzero value indicates failure. If a nonzero return code is provided by the exit, the z/OS XML parser saves it in the extended diagnostic area so that the caller of the parser has access to it by calling GXL1CTL (GXL4CTL).

Example

For an example of using this exit service, see "GXLSYM example" on page 231. These examples are located in SYS1.SAMPLIB .

Default Implementation

There is no default implementation. If this exit is not specified by the caller, StringIDs are not used by the z/OS XML parser. Length/value pairs representing all strings from the XML text are passed through to the parsed data stream for return to the caller. See "String Identifiers" on page 38 for more details about length/value pairs and StringIDs in the parsed data stream.

GXLSTRI — StringID service for Language Environment and Metal C

Description

This service provides a combination Language Environment and Metal C StringID service exit for the z/OS XML parser and the OSR generator.

Performance Implications

There are no performance implications.

Syntax

```
call gxlstri,(sys_svc_parm,
              string,
              str_len,
              stringID,
              ccsid,
              diag_code,
              return_code)
```

Parameters

sys_svc_parm
Supplied parameter

Type: Address

Length:
Fullword (Doubleword)

A pointer to the address of the storage to be used for this exit.

string
Supplied parameter

Type: Character string

Length:
determined by the str_len parameter

The string passed in from the OSR generator.

str_len
Supplied parameter

Type: Integer

Length:
Fullword

The value of the string length passed in from the OSR generator.

stringID
Returned parameter

Type: Unsigned integer

Length:
Fullword

The value of the StringID set by this exit.

ccsid
Supplied parameter

Type: Integer

Length:
Fullword

The Coded Character Set Identifier passed in from the OSR generator.

diag_code
Returned parameter

Type: Integer

Length:
Fullword

The diagnostic code set by this exit.

return_code
Returned parameter

Type: Integer

Length:
Fullword

The return code set by this exit.

Return Codes:

The z/OS XML parser uses the convention that the exit will provide a return code value of zero when successful. Any nonzero value indicates failure. If a nonzero return code is provided by the exit, the z/OS XML parser saves it in the extended diagnostic area so that the caller of the parser has access to it by calling GXL1CTL (GXL4CTL).

Example

For an example of using this exit service, see “GXLESTRI” on page 233.

Default Implementation

There is no default implementation.

Chapter 9. Diagnosis and problem determination

The diagnostic facilities of this z/OS XML parser can be used to debug both the operation of the z/OS XML parser itself and the input XML document. Since well-formedness checking is an integral part of the parsing process, and since the complexity of XML documents can be very high, the opportunity for encountering a flaw in the input stream that is difficult to diagnose is significant. To assist in diagnosis, the z/OS XML parser provides the following support:

- XMLDATA IPCS subcommand
- Diagnostic Area
- SLIP trap for reason codes from z/OS XML parser
- ARR recovery routine

XMLDATA IPCS subcommand

To make it easier to analyze z/OS XML System Services dumps, the **XMLDATA** subcommand is provided for use with the IPCS formatter. To use the subcommand, input the following under IPCS option 6:

```
COMMAND: XMLDATA address option
```

The *address* parameter is the address of the z/OS XML parser's Parser Anchor Block (PAB); this is a required parameter. The *address* parameter accepts both 31- and 64-bit addresses. If you do not know the value for the *address* parameter, you can place a '0' in the *address* field, and XMLDATA will try to locate the value for you, for example: `XMLDATA 0 TRACE`. Although this method is not guaranteed to work, it is still an available option.

The *option* parameter allows you to select what information you want to review within the provided dump (see Table 36 for a list of options and their descriptions). If nothing is provided for the *option* parameter, **XMLDATA** will use the default option BASIC. The following table lists the options available for **XMLDATA**:

Table 36. XMLDATA options

Option	Description
BASIC	Displays to the screen widely used dump information. Such information includes the following: the PSW and any general information during the abend; the value of the registers; an API trace; a user input parameter list; feature flags, return code and reason codes; and the last 64 bytes of the input and output buffers.
PARAM	Displays the parameter list values for the GXL1PRS or GXL4PRS entry points.

Table 36. XMLDATA options (continued)

Option	Description
BUFFER (*inlen*, *outlen*, *fraglen*)	Displays the last *inlen* bytes of the input buffer ending at where the parser abends, displays the last *outlen* bytes of the output buffer and displays the first *fraglen* bytes of the fragment buffer. The fragment buffer option is only available for a non-validating dump. For a validating dump, the input buffer option will not display the most current bytes of data at where the z/OS XML parser abends, but instead the input buffer option will display from the beginning of the input buffer for *inlen* bytes that has been loaded for parsing within the validating z/OS XML parser. If the length value of zero is provided for a specific buffer type, that specific buffer information will be skipped. The *inlen*, *outlen*, and *fraglen* parameters are all optional. For any that are not specified, the default is 128 bytes.
EXTENT	Displays all available free and external extents' information.
MISC	Displays the status of each feature flag, input document encoding, exit services, return code and reason codes.
TRACE (*option*)	Displays the trace of the API calls. The *option* parameter is optional. Providing 'ADV' in the *option* parameter displays a more advanced API trace. Otherwise, a simple API trace will be displayed. (Default is to display a simple API trace).
PAB	Displays all the defined fields in the PAB.
STRUCT (*option*, *address*)	Displays the formatted control blocks including the z/OS XML parser diagnostic area, element stack, default attribute record, local name tree, prefix tree, namespace tree and data buffer. The data buffer option is only available if the dump is taken with the validation feature flag turned on. The *option* parameter is required, otherwise no control block will be displayed. The options include the following: XD, XELE, XATT, LN, PFX, URI, DBUF, respectively. The *address* parameter is optional and is only available for local name, prefix, namespace tree and data buffer option. For local name, prefix, and namespace trees, if you do not want to display the tree from the root node, then provide a child node address for the tree to use as the root node. For data buffer, if you want to display the details of a specific internal input buffer, then provide a data buffer address. (For the address parameter, the default for the trees is the tree root node address.) If no options or addresses are selected, a menu of all available options will be displayed.
MARKED	Displays data that was parsed by the z/OS XML parser, but has not yet been placed in the output buffer, due to the interruption of an abend. This option is only available if the dump is taken with the validation feature flag turned off.
PMM	Displays the formatted Module Map: PMM, Secondary Table: PST, and System Control: PSC.
HELP	Displays all available options and their descriptions.

The following is an example of the **XMLDATA** subcommand:

```
XMLDATA 00002940121498028 PAB
```

Diagnostic Area

On the GXL1CTL (GXL4CTL) call, there is a diagnostic area where the z/OS XML parser places information that can be useful when debugging a failure or incorrect behavior in the parser. This area is mapped by macro GXLYXD. The diagnostic area contains the following fields:

XD_Eye
Eyecatcher GXLYXD

XD_Version
The z/OS XML parser version number.

XD_PAB
Address of Parser Anchor Block for this parse instance.

XD_InBuff
Address of current input buffer.

XD_InBuffOffset
Offset into input buffer where the z/OS XML parser stopped.

XD_OutBuff
Address of current output buffer.

XD_OutBuffOffset
Offset into output buffer where the last valid entry can be found.

XD_StorageRequested
Amount of storage that requested for request that failed.

XD_LastRC
Return code from the last call to GXP1PRS (GXP4PRS).

XD_LastRSN
Reason code from the last call to GXP1PRS (GXP4PRS).

XD_StorageRC
Return code from call to STORAGE.

XD_StorageRsn
Reason code from call to STORAGE.

XD_Iarv64Rc
Return code from call to IARV64.

XD_Iarv64Rsn
Reason code from call to IARV64.

XD_StorExitRc
Return code from storage exit.

XD_StorExitRsn
Reason code from storage exit.

XD_StorExitDiag
The diagnostic code from the storage exit.

XD_SymExitRc
Return code from symbol exit.

XD_SymExitDiag
The diagnostic code from symbol exit.

XD_SymbolLength
Length of the symbol which was rejected by the user symbol exit routine.

XD_IFA_RC
The return code from the request to run on a zAAP.

XD_EndOfDocRC
Return code from a finished parse.

XD_EndOfDocRSN
Reason code from a finished parse.

XD_MIN_OB
Minimum output buffer size required on next parser call.

XD_LastOutput
Output buffer area in PIMA containing enhanced error records.

SLIP trap for return codes from the z/OS XML parser

To obtain a dump on a specific reason code from any of the z/OS XML parser callable services, use the release appropriate SLIP example in the following table:

Table 37. SLIP examples by release

z/OS release	SLIP example

V1.11 or higher	`SLIP SET,IF,A=SYNCSVCD,RANGE=(10?+220?+48?+8?+E),`
	`DATA=(4G!+F0!+b2,EQ,xxxx),`
	`SDATA=(CSA,LPA,TRT,SQA,RGN,SUM),j=jobname,END`

where `xxxx` is one of the 4 digit (2 byte) reason codes listed in Appendix B, “Reason codes listed by value,” on page 161 that is to be trapped and `j=jobname` is the optional jobname that is expected to issue the error (for example, `j=IBMUSER`).

ARR recovery routine

z/OS XML provides an ARR recovery routine to assist with problem determination and diagnostics. This recovery routine can be turned on through an initialization option when invoked through the assembler API. For callers of the C/C++ parser API (gxlpParse), when running in Language Environment, the ARR recovery routine is provided by default in most cases. For C or C++ callers who are running in either SRB mode or under an existing FRR routine, the z/OS XML ARR will not be provided, as it would not work properly in those environments.

If the z/OS XML parser abends, the z/OS XML ARR routine will get control and will collect dumps and return to the caller with a XRC_FATAL return code. For unauthorized callers, an IEATDUMP will be taken in data set *userid*.GXLSCXML.DYYMMDD.THHMMSS.DUMP, where DYYMMDD is the date and THHMMSS is the time the dump was taken. The task level ACEE is used to obtain the *userid*. If there is no task level ACEE, the address space level ACEE is used. If there is no address space level ACEE, a dump is not taken. For authorized callers, an SDUMPX will be taken into a system dump data set.

If the user would like to continue parsing, he must terminate and re-initialize a PIMA following any abend in the z/OS XML parser.

Appendix A. Return codes listed by value

This section lists return codes by value and describes them.

Hex Value	Return Code	Description
0000	XRC_SUCCESS	The z/OS XML parser service was successful.
0004	XRC_WARNING	The z/OS XML parser service has partial success.
0008	XRC_FAILURE	Processing failed. Returned data areas and parms valid.
000C	XRC_NOT_WELL_FORMED	The document is not-well-formed.
0010	XRC_FATAL	Processing failed. Returned data areas or output parameters cannot be relied on to contain valid data.
0014	XRC_LOAD_FAILED	The load of the specified service failed. The return code from the LOAD macro is returned in the reason code field.
0018	XRC_NOT_VALID	The document is not valid according to the specified schema.

Appendix B. Reason codes listed by value

This section describes reason codes, listing them by hexadecimal value and describing actions to correct the error.

Reason code value

0000 **XRSN_SUCCESS**

The z/OS XML parser service was successful.

Action: None

1000 **XRSN_PIMA_NOT_INITIALIZED**

The PIMA passed to a z/OS XML parser service is unusable.

Action: The PIMA passed has not been initialized with a call to the z/OS XML parser initialization service GXL1INI or GXL4INI or the PIMA address is incorrect.

1001 **XRSN_PIMA_SMALL**

The length of the PIMA is too small.

Action: The size of the PIMA passed on GXL1INI or GXL4INI must be at least the minimum required size for the requested features. Refer to the z/OS XML User's Guide for the correct minimum value.

1002 **XRSN_PIMA_RESIDUAL_DATA**

Initialization has already been done on this PIMA.

Action: The GXL1INI or GXL4INI service has been called to initialize the PIMA, but the PIMA storage has already been initialized. You must call GXL1TRM or GXL4TRM before the PIMA can be reinitialized to guarantee that all resources have been cleaned up.

1004 **XRSN_PIMA_INCONSISTENT_STATE**

The z/OS XML parser exited without cleaning up.

Action: Attempt to collect a dump of the problem. The joblog for the address space should contain a symptom dump which identifies the abend code. If running from a user address space, allocate a SYSMDUMP DD and recreate the problem. If running in some system address space, use SLIP to get a dump of the abend. Contact your system administrator for help in getting the dump and possibly contacting IBM.

1005 **XRSN_CTL_DATA_PARM_INVALID**

The CTL_DATA parm is invalid.

Action: It is null, but is a required input parmameter for this feature flag. Call the ctl function again, passing in the required parameter.

1006 **XRSN_IMODV_NOT_LOADED**

The validating parser has not been loaded.

Action: Invoke the GXL1LOD or GXL4LOD to load the validating parser. Call initialization again, after a successful load.

Reason code value

1007 **XRSN_CTL_DATA_VERSION_INVALID**

The input control block version is invalid.

Action: The version field in the input control block is set to an invalid value.

1008 **XRSN_CTL_XEAR_RC_INVALID**

The XEAR_REPLACEMENT_CHAR_LENGTH is invalid.

Action: The XEAR_REPLACEMENT_CHAR_LENGTH field is set to an invalid value. It must be set to one.

1100 **XRSN_STORAGE_31_GET_ERROR**

Unable to allocate memory.

Action: If your application does not already call GXL1CTL after the parse, add a call to GXL1CTL. The address returned by GXL1CTL points to an area mapped by GXLYXD. Extract the return and reason code from the XD area, pertaining to storage access failures that occurred using the STORAGE macro. Contact your system administrator for help in interpreting these values.

1101 **XRSN_STORAGE_64_GET_ERROR**

Unable to allocate memory.

Action: If your application does not already call GXL1CTL after the parse, add a call to GXL1CTL. The address returned by GXL1CTL points to an area mapped by GXLYXD. Extract the return and reason code from the XD area, pertaining to storage access failures using the IARV64 service. Contact your system administrator for help in interpreting these values.

1140 **XRSN_STORAGE_GET_EXIT_TOO_SMALL**

The storage returned from get storage exit is too small.

Action: If your application does not already call GXL1CTL after the parse, add a call to GXL1CTL. The address returned by GXL1CTL points to an area mapped by GXLYXD. Extract the return and reason code from the XD area, pertaining to storage exit failure. Contact your system administrator for help in interpreting these values.

1143 **XRSN_STORAGE_31_SFREE_ERROR**

Single failure when attempting to free storage.

Action: Contact your system administrator.

1144 **XRSN_STORAGE_31_MFREE_ERROR**

Multiple failures when attempting to free storage.

Action: Contact your system administrator.

1145 **XRSN_STORAGE_64_SFREE_ERROR**

Single failure when attempting to free storage.

Action: Contact your system administrator.

Reason code value	
1146	**XRSN_STORAGE_64_MFREE_ERROR** Multiple failures when attempting to free storage. *Action:* Contact your system administrator.
1147	**XRSN_STORAGE_CORRUPTED_ERROR** Storage header has been corrupted. *Action:* Contact your system administrator.
1148	**XRSN_INPUT_BUFFER_ACCESS_ERROR** The user abended when trying to access the input buffer. *Action:* Check the input buffer parameter and length passed into the parser to be sure they are correct. If the input parameters are correct, Contact your system administrator. .
1149	**XRSN_INPUT_BUFFER_ACCESS_ERROR_ND** The user abended when trying to access the input buffer. No dump was taken. *Action:* Check the input buffer parameter and length passed into the parser to be sure they are correct. If the input parameters are correct, Contact your system administrator. .
1150	**XRSN_OUTPUT_BUFFER_ACCESS_ERROR** The user abended when trying to access the output buffer. *Action:* Check the output buffer parameter and length passed into the parser to be sure they are correct. If the output parameters are correct, Contact your system administrator. .
1151	**XRSN_OUTPUT_BUFFER_ACCESS_ERROR_ND** The user abended when trying to access the output buffer. No dump was taken. *Action:* Check the output buffer parameter and length passed into the parser to be sure they are correct. If the output parameters are correct, Contact your system administrator. .
1152	**XRSN_PIMA_ACCESS_ERROR** The user abended when trying to access the PIMA. *Action:* Check the PIMA parameter and length passed into the parser to be sure they are correct. If the PIMA parameters are correct, Contact your system administrator. .
1153	**XRSN_PIMA_ACCESS_ERROR_ND** The user abended when trying to access the PIMA. No dump was taken. *Action:* Check the PIMA parameter and length passed into the parser to be sure they are correct. If the PIMA parameters are correct, Contact your system administrator. .

Reason code value	
1154	**XRSN_UNKNOWN_ERROR** An unknown abend occurred. *Action:* Contact your system administrator.
1155	**XRSN_UNKNOWN_ERROR_ND** Unknown abend occurred and no dump was taken. *Action:* Contact your system administrator.
1156	**XRSN_STORAGE_OBTAIN_FAILED** A storage obtain request failed *Action:* Contact your system administrator.
1157	**XRSN_STORAGE_OBTAIN_FAILED_ND** A storage obtain request failed, no dump taken *Action:* Contact your system administrator.
1201	**XRSN_PARM_ENCODING_SPEC_INVALID** The ccsid passed is not supported. *Action:* The CCSID parameter on the call to GXL1INI or GXL4INI is not one of the supported character encodings. Pass only permitted CCSID parameters. See the documentation of the GXL1INI service for supported ccsid constants.
1202	**XRSN_PARM_FEATURE_FLAG_INVALID** Undefined feature flag is set *Action:* The feature flag parameter passed to GXL1INI or GXL4INI or GXL1CTL or GXL4CTL has an undefined bit set or a bit that is invalid for this api set. You can only set features that are defined or supported on the api.
1203	**XRSN_PARM_UNSUPPORT_ENCODING** XML encoding string is not supported. *Action:* The encoding string in the XML declaration is not supported. Use only the supported encoding names.
1204	**XRSN_OPERATION_FLAG_INVALID** Undefined operation flag is set. *Action:* The operation flag is set to an invalid value.
1300	**XRSN_BUFFER_INBUF_SMALL** The input buffer size is too small. *Action:* The query service was not able to parse a complete XML declaration. The caller needs to pass more of the document to the service.

Reason code value	
1301	**XRSN_BUFFER_INBUF_END** The end of the input buffer has been reached. *Action:* This is a normal reason code for spanning buffers.
1302	**XRSN_BUFFER_OUTBUF_SMALL** The output buffer was too small to contain the next item. *Action:* The caller must reset the parser, then parse the document again from the beginning, passing in a larger output buffer.
1303	**XRSN_BUFFER_OUTBUF_END** The end of the output buffer has been reached *Action:* This is a normal reason code for spanning buffers.
1304	**XRSN_BUFFER_INOUTBUF_END** The end of both buffers have been reached *Action:* This is a normal reason code for spanning buffers.
1305	**XRSN_STORAGE_GET_EXIT_ERROR** Application storage exit unable to allocate memory. *Action:* If your application does not already call GXL1CTL after the parse, add a call to GXL1CTL. The address returned by GXL1CTL points to an area mapped by GXLYXD. Extract the return and reason code from the XD area, pertaining to storage access failures. Contact your system administrator for help in interpreting these values.
1307	**XRSN_STORAGE_SFREE_EXIT_ERROR** User free storage exit has one failure. *Action:* Contact your system administrator.
1308	**XRSN_STORAGE_MFREE_EXIT_ERROR** User free storage exit has multiple failures. *Action:* Contact your system administrator.
1309	**XRSN_DYNAMIC_CODE_CHANGE** z/OS XML parser was re-installed. *Action:* Caller needs to terminate the parser and restart with parser initialization.
1310	**XRSN_SYM_EXIT_ERROR** The symbol exit returned an error. *Action:* Contact the owner of the symbol exit and have them debug the problem.

Reason code value	
1400	**XRSN_DEALLOC_EXIT_MISSING** Allocation exit specified without deallocation exit *Action:* The service exit specification on a call to GXL1INI or GXL4INI contains an exit to allocate storage, but no exit to deallocate storage. Either both or neither is required.
1401	**XRSN_ALLOC_EXIT_MISSING** Deallocation exit specified without allocation exit *Action:* The service exit specification on a call to GXL1INI or GXL4INI contains an exit to deallocate storage, but no exit to allocate storage. Either both or neither is required.
1403	**XRSN_OPTN_UNKNOWN** Unsupported value set on the options parameter. *Action:* Refer to the API documentation for the correct values to pass to this service.
1404	**XRSN_QXDWORK_AREA_SMALL** Query service work area length is too small. *Action:* Pass a bigger area.
1405	**XRSN_INTERNAL_ERROR** Internal error in the z/OS XML parser. *Action:* Contact your system administrator.
1407	**XRSN_FEATURE_FLAG_INVALID_IN_ENV** The recovery feature flag is on, but the program either has an existing FRR or is in SRB mode. This feature is not valid in these environments. *Action:* Reinitialize the parse with the recovery feature flag turned off.
1408	**XRSN_INVALID_OPTION** The operation being performed is not valid for this service. *Action:* Refer to the API documentation to determine which parsing services this option is valid for.
1500	**XRSN_SVC_UNKNOWN** The code specified for the svc_code parameter is invalid. *Action:* Refer to the API documentation for the correct values for the svc_code parameter.
1501	**XRSN_NO_OSR_SPECIFIED** No OSR has been loaded via a CTL call. *Action:* Perform a CTL_LOAD_OSR operation via CTL with a nonzero XOSR_OSR_PTR.

Reason code value	
1502	**XRSN_NO_SCHEMAS_SPECIFIED** Either the schema vector parameter passed was NULL, or the number of schemas specified in the vector was zero. *Action:* Pass in a valid schema vector that contains one or more text. schemas to process.
1503	**XRSN_NO_OSR_BUFFER_SPECIFIED** No OSR buffer was for generation. *Action:* Pass in the address of a buffer to receive a generated OSR.
1504	**XRSN_OSR_INVALID** The data within the OSR is invalid. *Action:* Ensure that the correct address of the OSR is being passed.
1505	**XRSN_NEED_OSR** All schema location information has been returned from the instance document. A LOAD_OSR operation may be necessary to validate this document. *Action:* If an OSR has been loaded and can be used to validate the instance document, no special action is necessary. Otherwise, load an OSR to validate this document.
1506	**XRSN_NO_FRAGPATH_SPECIFIED** No fragment path has been loaded via a CTL call. *Action:* Perform a load fragment context operation via CTL with a fragment path.
1508	**XRSN_CTL_FRAGPATH_INCORRECT** The provided fragment path is incorrect. *Action:* Change the Fragment Path to correct the error and retry.
1509	**XRSN_OSR_INCOMPATIBLE** The OSR is incompatible with the specified feature *Action:* Change the document or schema to correct and retry.
1510	**XRSN_XRR_INVALID** The data within the XRR is invalid. *Action:* Ensure that the correct address of the XRR is being passed.
1511	**XRSN_CTL_FRAG_PREV_ENABLED** Document fragment parsing is already enabled. Issuing this control call is not allowed. *Action:* Please disable fragment parsing and retry.

Reason code value	
1512	**XRSN_CTL_FRAG_PREV_DISABLED** Document fragment parsing is already disabled. Issuing this control call is not allowed. *Action:* Please enable fragment parsing and retry.
1513	**XRSN_CTL_SEQUENCE_INCORRECT** This control call cannot be issued under the present parse conditions. *Action:* Correct the sequence of calls and retry.
1514	**XRSN_CTL_FRAG_NSCONTEXT_INCORRECT** The provided fragment NS context is incorrect. *Action:* Change the Fragment namespace context to correct the error and retry.
1515	**XRSN_CTL_FRAGPATH_ROOT_RESTRICTED** The fragment path root element is invalid. *Action:* The provided fragment path's root element does not match with the Restricted Root Elements. Correct the error and retry.
1516	**XRSN_CTL_XDBX_NO_ENTITIES** No entities are present in XDBX streams. *Action:* XDBX input streams will not contain any entity references. The entities-and-references operation has no effect in this case.
2000	**XRSN_COMMENT_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within comment markup. *Action:* Change the document to correct the error and retry.
2001	**XRSN_CDATA_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within CDATA markup. *Action:* Change the document to correct the error and retry.
2002	**XRSN_PI_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within processing instruction markup. *Action:* Change the document to correct the error and retry.
2003	**XRSN_ATTR_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within attribute markup. *Action:* Change the document to correct the error and retry.

Reason code value	
2004	**XRSN_ENDTAG_NOT_REACHED** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended without reaching the document element end tag. *Action:* Change the document to correct the error and retry.
2006	**XRSN_TAG_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within an element start tag. *Action:* Change the document to correct the error and retry.
2007	**XRSN_NS_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within namespace declaration markup. *Action:* Change the document to correct the error and retry.
2008	**XRSN_XML_DECL_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within the XML declaration. *Action:* Change the document to correct the error and retry.
2009	**XRSN_DTD_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within doctype declaration markup. *Action:* Change the document to correct the error and retry.
2010	**XRSN_SUBSET_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within internal subset markup. *Action:* Change the document to correct the error and retry.
2011	**XRSN_SUBSET_ELEM_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within an element declaration. *Action:* Change the document to correct the error and retry.
2012	**XRSN_SUBSET_NOTATION_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within a notation declaration. *Action:* Change the document to correct the error and retry.

Reason code value	
2013	**XRSN_SUBSET_COMMENT_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within comment markup. *Action:* Change the document to correct the error and retry.
2015	**XRSN_SUBSET_PEREF_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within a parameter entity reference. *Action:* Change the document to correct the error and retry.
2016	**XRSN_SUBSET_ENTITY_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within an entity declaration. *Action:* Change the document to correct the error and retry.
2017	**XRSN_SUBSET_ATTL_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within an attribute list declaration. *Action:* Change the document to correct the error and retry.
2018	**XRSN_MARKUP_INCOMPLETE** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended within markup. *Action:* Change the document to correct the error and retry.
2019	**XRSN_DOC_ELEM_NOT_FOUND** The GXL1CTL (GXL4CTL) API was called with the finish option and the input document was not complete. The document ended without finding the document element. *Action:* Change the document to correct the error and retry.
2020	**XRSN_LENGTH_VALUE_INVALID** The length value is incorrect because the upper most bit of a length variable's value is not zero, and the variable type is defined as 31 bit. *Action:* Correct the length value and retry.
2021	**XRSN_FRAGMENT_INVALID** The parsed document fragment is incorrect. *Action:* Change the document fragment to correct the error and retry.
2022	**XRSN_DOCUMENT_INVALID** The parsed document is incorrect. *Action:* Change the document to correct the error and retry.

Reason code value	
2024	**XRSN_PREV_OUTBUF_PENDING** The parsed data is pending for output. *Action:* Parse the document again with the neccessary output buffer.
3000	**XRSN_ATTR_DUPLICATE** Duplicate attributes were found. *Action:* Change the document to correct the error and retry.
3001	**XRSN_NS_DUPLICATE** Duplicate namespace declaration found. *Action:* Change the document to correct the error and retry.
3002	**XRSN_NS_ATTR_PREFIX_NOT_DECL** Namespace prefix on attribute not declared. *Action:* Change the document to correct the error and retry.
3003	**XRSN_NS_ELEM_PREFIX_NOT_DECL** Namespace prefix on element tag not declared. *Action:* Change the document to correct the error and retry.
3004	**XRSN_ENC_DETECTED_INVALID** Encoding detected during query is unsupported. *Action:* During the query service, an unsupported byte sequence is found at the beginning of the document.
3006	**XRSN_CHAR_ERROR** Incorrectly encoded character found in the input stream. *Action:* Contact your system administrator.
3007	**XRSN_COMMENT_DASH_MISSING** Comment without starting dash found. *Action:* Check the document for a comment markup missing a dash in the beginning and correct the document.
3008	**XRSN_COMMENT_CHAR_INVALID** Comment markup contains incorrect character. *Action:* Change the document to correct the error and retry.
3009	**XRSN_COMMENT_RIGHT_ANGLE_MISSING** Comment is missing the ending angle bracket at the end of the markup. *Action:* Change the document to correct the error and retry.
3010	**XRSN_CDATA_KEYWORD_INVALID** CDATA keyword expected but not found. *Action:* Change the document to correct the error and retry.

Reason code value	
3011	**XRSN_CDATA_LEFT_BRACKET_MISSING** Left square bracket expected in CDATA markup. *Action:* Change the document to correct the error and retry.
3013	**XRSN_CDATA_CHAR_INVALID** A character was found that is not allowed within a CDATA section. *Action:* Change the document to correct the error and retry.
3017	**XRSN_PI_CHAR_INVALID** A character was found that is not allowed within a Processing Instruction. *Action:* Change the document to correct the error and retry.
3018	**XRSN_ATTR_NAME_CHAR_INVALID** A character was found that is not allowed within an attribute name. *Action:* Change the document to correct the error and retry.
3019	**XRSN_ATTR_LNAME_CHAR_INVALID** A character was found that is not allowed within an attribute local name. *Action:* Change the document to correct the error and retry.
3020	**XRSN_ATTR_EQUAL_MISSING** An incorrect character was found after the attribute name, and the only character allowed is "=". *Action:* Change the document to correct the error and retry.
3021	**XRSN_ATTR_QUOTE_MISSING** An incorrect character was found after the attribute "=" character, and the only characters allowed here is either white space, or a single or double quote. *Action:* Change the document to correct the error and retry.
3022	**XRSN_ATTR_VALUE_CHAR_INVALID** An incorrect character was found in an attribute value. *Action:* Change the document to correct the error and retry.
3023	**XRSN_ATTR_REF_CHAR_INVALID** An incorrect character was found in entity reference in an attribute value. *Action:* Change the document to correct the error and retry.
3024	**XRSN_ATTR_REF_NAME_CHAR_INVALID** An incorrect character was found in entity reference in an attribute value. *Action:* Change the document to correct the error and retry.

Reason code value	
3025	**XRSN_ATTR_REF_VALUE_INVALID** Incorrect character found in character entity reference in an attribute value. *Action:* Change the document to correct the error and retry.
3026	**XRSN_CONTNT_REF_CHAR_INVALID** An incorrect character was found in entity reference in element content. *Action:* Change the document to correct the error and retry.
3027	**XRSN_CONTNT_REF_NAME_INVALID** An incorrect character was found in entity reference in element content. *Action:* Change the document to correct the error and retry.
3028	**XRSN_CONTNT_REF_VALUE_INVALID** An incorrect character was found in character entity reference in element content. *Action:* Change the document to correct the error and retry.
3029	**XRSN_MARKUP_INVALID** An incorrect character is found within markup. *Action:* Change the document to correct the error and retry.
3030	**XRSN_CONTNT_CHAR_INVALID** An incorrect character is found in element content *Action:* Change the document to correct the error and retry.
3031	**XRSN_TAG_ELEMNAME_INVALID** An incorrect character is found in an element tag name *Action:* Change the document to correct the error and retry.
3032	**XRSN_TAG_LNAME_INVALID** An incorrect character is found in an element tag name. *Action:* Change the document to correct the error and retry.
3033	**XRSN_TAG_CHAR_INVALID** An incorrect character is found in an element start tag. *Action:* Change the document to correct the error and retry.
3034	**XRSN_TAG_EMPTY_INVALID** An incorrect character is found after the "/" character to end the element tag. The only character allowed is a greater than symbol to end the empty element tag. *Action:* Change the document to correct the error and retry.

Reason code value	
3035	**XRSN_ENDTAG_NAME_MISMATCH** At the element end tag, a mis-match element name is found compared to the name of the start element *Action:* Change the document to correct the error and retry.
3036	**XRSN_ENDTAG_EMPTY_TAG_INVALID** An incorrect character is found in the element end tag after the element name. The only characters allowed after the name is white space or the greater than symbol. *Action:* Change the document to correct the error and retry.
3038	**XRSN_NS_CHAR_INVALID** Incorrect character found in namespace URI. *Action:* Change the document to correct the error and retry.
3039	**XRSN_NS_WHITESPACE_CHAR_INVALID** Incorrect character in namespace declaration. Expecting either white space or "=". *Action:* Change the document to correct the error and retry.
3040	**XRSN_NS_PFX_NAME_INVALID** An incorrect character is found in the prefix name portion of a namespace declaration. *Action:* Change the document to correct the error and retry.
3041	**XRSN_NS_QUOTE_MISSING** Incorrect character in namespace declaration after the "=" character. Expected a single or double quote or a white space character. *Action:* Change the document to correct the error and retry.
3042	**XRSN_NS_REF_CHAR_INVALID** An incorrect character was found in entity reference in a namespace declaration. *Action:* Change the document to correct the error and retry.
3043	**XRSN_NS_REF_NAME_CHAR_INVALID** An incorrect character was found in entity reference in a namespace declaration. *Action:* Change the document to correct the error and retry.
3044	**XRSN_NS_REF_VALUE_INVALID** Incorrect character found in character entity reference in a namespace declaration. *Action:* Change the document to correct the error and retry.

Reason code value	
3045	**XRSN_DTD_DOCTYPE_INVALID** Incorrect character found while parsing DOCTYPE keyword. *Action:* Change the document to correct the error and retry.
3046	**XRSN_XML_VER_VALUE_INVALID** An incorrect XML version number was specified. The only allowed values are "1.0" or "1.1". *Action:* Change the document to correct the error and retry.
3047	**XRSN_XML_VER_KEYWORD_INVALID** The characters do not match the word "version" *Action:* Change the document to correct the error and retry.
3048	**XRSN_XML_VER_EQUAL_MISSING** Expected white space or "=" character after "version". *Action:* Change the document to correct the error and retry.
3049	**XRSN_XML_VER_QUOTE_MISSING** An incorrect character is detected after the "=" where it is expected to be a single quote, double quote or a white space character. *Action:* Change the document to correct the error and retry.
3050	**XRSN_XML_CHAR_INVALID** In the XML Declaration after the close of the version value, an incorrect character is detected. *Action:* Change the document to correct the error and retry.
3051	**XRSN_XML_NAME_CHAR_INVALID** Incorrect character in XML Declaration. Expected either "s" for standalone, "e" for encoding, white space or "?". *Action:* Change the document to correct the error and retry.
3052	**XRSN_XML_ENC_KEYWORD_INVALID** The characters do not match the word "encoding". *Action:* Change the document to correct the error and retry.
3053	**XRSN_XML_ENC_EQUAL_MISSING** Expected white space or "=" character after "encoding". *Action:* Change the document to correct the error and retry.
3054	**XRSN_XML_ENC_QUOTE_MISSING** An incorrect character is detected after the "=" where it is expected to be a single quote, double quote or a white space character. *Action:* Change the document to correct the error and retry.

Reason code value	
3055	**XRSN_XML_ENC_CHAR_INVALID** An incorrect character is detected in the XML Declaration encoding value. *Action:* Change the document to correct the error and retry.
3056	**XRSN_XML_STD_KEYWORD_INVALID** The characters do not match the word "standalone" *Action:* Change the document to correct the error and retry.
3057	**XRSN_XML_STD_VALUE_INVALID** An incorrect value for standalone was specified. The only allowed values are "yes" or "no". *Action:* Change the document to correct the error and retry.
3058	**XRSN_XML_STD_EQUAL_MISSING** Expected white space or "=" character after "standalone". *Action:* Change the document to correct the error and retry.
3059	**XRSN_XML_STD_QUOTE_MISSING** An incorrect character is detected after the "=" where it is expected to be a single quote, double quote or a white space character. *Action:* Change the document to correct the error and retry.
3060	**XRSN_XML_END_CHAR_INVALID** An incorrect character is detected at the end of the XML declaration, where "?>" is expected. *Action:* Change the document to correct the error and retry.
3061	**XRSN_ENTITY_NOT_DEFINED** Entity not defined or not defined correctly. *Action:* Change the document to correct the error and retry.
3062	**XRSN_CHAR_INVALID** An incorrect character was detected in the document. Either white space or "<" was expected. *Action:* Change the document to correct the error and retry.
3063	**XRSN_PROLOGUE_CHAR_INVALID** The initial character in the document was incorrect. Either white space or "<" was expected. Possibly the document encoding does not match the parser encoding specified during initialization. *Action:* Change the document to correct the error and retry.
3064	**XRSN_XML_DECL_NOT_ALLOWED** Any Characters other than the Byte Order Mark (BOM) are not allowed before the XML declaration in the XML document. *Action:* Change the document to correct the error and retry.

Reason code value	
3065	**XRSN_MULTIPLE_DOC_ELEMENTS** Multiple elements were found at the document level. Only one is allowed. *Action:* Change the document to correct the error and retry.
3066	**XRSN_ENTITY_LOOP_REF** An entity refers directly, or indirectly to itself. Recursion is not allowed. *Action:* Change the document to correct the error and retry.
3067	**XRSN_NS_URI_EMPTY** A non-default namespace declaration contains a URI value of zero length and the XML version is 1.0. *Action:* Change the document to correct the error and retry.
3068	**XRSN_INVALID_CHAR_SEQ** An invalid character sequence found in the content portion of the document. *Action:* Change the document to correct the error and retry.
3069	**XRSN_ENTITY_MARKUP_INCOMPLETE** Incomplete markup in entity. *Action:* Change the document to correct the error and retry.
3070	**XRSN_TEXT_DECL_INCOMPLETE** The text declaration markup is not well-formed. The document ended within the text declaration. *Action:* Change the document to correct the error and retry.
3071	**XRSN_TEXT_VER_VALUE_INVALID** An incorrect version number was specified in the text declaration. The only allowed values are "1.0" or "1.1". *Action:* Change the document to correct the error and retry.
3072	**XRSN_TEXT_VER_KEYWORD_INVALID** The characters do not match the word "version" *Action:* Change the document to correct the error and retry.
3073	**XRSN_TEXT_VER_EQUAL_MISSING** Expected white space or "=" character after "version" *Action:* Change the document to correct the error and retry.
3074	**XRSN_TEXT_VER_QUOTE_MISSING** An incorrect character is detected after the "=" where it is expected to be a single quote, double quote or a white space character. *Action:* Change the document to correct the error and retry.

Reason code value	
3075	**XRSN_TEXT_CHAR_INVALID** Incorrect character detected after the close of the version value in the text declaration. *Action:* Change the document to correct the error and retry.
3076	**XRSN_TEXT_NAME_CHAR_INVALID** Incorrect character in text declaration. Expected either "e" for encoding, white space or "?". *Action:* Change the document to correct the error and retry.
3077	**XRSN_TEXT_ENC_KEYWORD_INVALID** The characters do not match the word "encoding". *Action:* Change the document to correct the error and retry.
3078	**XRSN_TEXT_ENC_EQUAL_MISSING** Expected white space or "=" character after "encoding". *Action:* Change the document to correct the error and retry.
3079	**XRSN_TEXT_ENC_QUOTE_MISSING** An incorrect character is detected after the "=" where it is expected to be a single quote, double quote or a white space character. *Action:* Change the document to correct the error and retry.
3080	**XRSN_TEXT_ENC_CHAR_INVALID** An incorrect character is detected in the text declaration encoding value. *Action:* Change the document to correct the error and retry.
3081	**XRSN_TEXT_END_CHAR_INVALID** An incorrect character is detected at the end of the text declaration, where "?>" is expected. *Action:* Change the document to correct the error and retry.
3082	**XRSN_TEXT_DECL_NOT_ALLOWED** text declaration is only allowed in the beginning of each fragment scope defined by start and end fragment control operation. *Action:* Change the document to correct the error and retry.
3085	**XRSN_ENTITY_UNRESOLVABLE** Entity references in document fragment cannot be resolved. *Action:* Provide the necessary entities and retry.
5000	**XRSN_DTD_NAME_CHAR_INVALID** An incorrect character is detected after the root element name of the document type declaration where only "SYSTEM", "PUBLIC", square bracket, or greater than characters are allowed. *Action:* Change the document to correct the error and retry.

Reason code value	
5001	**XRSN_DTD_CHAR_INVALID** Incorrect character found in document type declaration. *Action:* Change the document to correct the error and retry.
5002	**XRSN_DTD_EXTERNALID_INVALID** The external ID keyword does not match the word "SYSTEM" or "PUBLIC". *Action:* Change the document to correct the error and retry.
5003	**XRSN_DTD_QUOTE_MISSING** Incorrect quotation delimiter after external identifier. It is expected to be a single quote, double quote or a white space character. *Action:* Change the document to correct the error and retry.
5004	**XRSN_DTD_FILENAME_INVALID** Incorrect character in external identifier filename. *Action:* Change the document to correct the error and retry.
5005	**XRSN_SUBSET_CHAR_INVALID** Incorrect character in internal subset of the DTD. *Action:* Change the document to correct the error and retry.
5006	**XRSN_SUBSET_MARKUP_INVALID** An incorrect character is detected within the markup keyword in the internal subset of the doctype declaration. *Action:* Change the document to correct the error and retry.
5007	**XRSN_ELEM_CONTNT_CHAR_INVALID** An incorrect character is found in the element content portion of the element type declaration located in the internal subset of the doctype declaration. *Action:* Change the document to correct the error and retry.
5008	**XRSN_ELEM_CHAR_INVALID** Incorrect character in element declaration in DTD. *Action:* Change the document to correct the error and retry.
5009	**XRSN_ELEM_LNAME_INVALID** An incorrect character is found in the element name portion of an element declaration. *Action:* Change the document to correct the error and retry.
5010	**XRSN_ELEM_ELEMNAME_INVALID** An incorrect character is found in the element name portion of an element declaration. *Action:* Change the document to correct the error and retry.

Reason code value	
5011	**XRSN_NTTN_CHAR_INVALID** Incorrect character in notation declaration in DTD. *Action:* Change the document to correct the error and retry.
5012	**XRSN_NTTN_NAME_INVALID** An incorrect character is found in the notation declaration name. *Action:* Change the document to correct the error and retry.
5013	**XRSN_NTTN_ID_INVALID** The external or public identifier string in the notation declaration does not match with the word "SYSTEM" or "PUBLIC". *Action:* Change the document to correct the error and retry.
5014	**XRSN_NTTN_QUOTE_MISSING** Incorrect quotation delimiter after external identifier. It is expected to be a single quote, double quote or a white space character. *Action:* Change the document to correct the error and retry.
5015	**XRSN_NTTN_FILENAME_INVALID** Incorrect character in notation identifier literal. *Action:* Change the document to correct the error and retry.
5020	**XRSN_PEREF_NAME_CHAR_INVALID** Incorrect character in parameter entity reference in DTD. *Action:* Change the document to correct the error and retry.
5021	**XRSN_ENTY_NAME_CHAR_INVALID** Incorrect character in entity declaration name in DTD. *Action:* Change the document to correct the error and retry.
5022	**XRSN_ENTY_CHAR_INVALID** Incorrect character in entity declaration in DTD. *Action:* Change the document to correct the error and retry.
5023	**XRSN_ENTY_VALUE_INVALID** Incorrect character in entity declaration value in DTD. *Action:* Change the document to correct the error and retry.
5024	**XRSN_ENTY_REF_CHAR_INVALID** An incorrect character was found in entity reference in an entity declaration. *Action:* Change the document to correct the error and retry.

Reason code value	
5025	**XRSN_ENTY_REF_NAME_INVALID** Incorrect character was found in entity reference in an entity declaration. *Action:* Change the document to correct the error and retry.
5026	**XRSN_ENTY_REF_VALUE_INVALID** Incorrect character found in character entity reference in an entity declaration. *Action:* Change the document to correct the error and retry.
5027	**XRSN_ENTY_QUOTE_MISSING** Incorrect quotation delimiter in entity declaration in DTD. It is expected to be a single quote, double quote or a white space character. *Action:* Change the document to correct the error and retry.
5028	**XRSN_ENTY_EXTERNALID_INVALID** The external or public identifier string in the entity declaration does not match with the word "SYSTEM" or "PUBLIC". *Action:* Change the document to correct the error and retry.
5029	**XRSN_ENTY_FILENAME_INVALID** Incorrect character in entity identifier value. *Action:* Change the document to correct the error and retry.
5030	**XRSN_ENTY_NDATA_INVALID** Incorrect character in entity NDATA declaration in DTD. *Action:* Change the document to correct the error and retry.
5031	**XRSN_ENTY_NDATA_NAME_INVALID** An incorrect character is found in the entity NDATA declaration name. *Action:* Change the document to correct the error and retry.
5040	**XRSN_ATTL_ELEMNAME_INVALID** An incorrect character is found in the attribute list declaration element name in the DTD. *Action:* Change the document to correct the error and retry.
5041	**XRSN_ATTL_CHAR_INVALID** An incorrect character is found in the attribute list declaration in the DTD. *Action:* Change the document to correct the error and retry.
5042	**XRSN_ATTL_NAME_CHAR_INVALID** An incorrect character is found in the attribute list declaration attribute name in the DTD. *Action:* Change the document to correct the error and retry.

Reason code value	
5043	**XRSN_ATTL_LNAME_CHAR_INVALID** An incorrect character is found in the attribute list declaration attribute name in the DTD. *Action:* Change the document to correct the error and retry.
5044	**XRSN_ATTL_TYPE_INVALID** Incorrect character in attribute list declaration type. The type must match one of these strings: "ID","IDREF","IDREFS","ENTITY","ENTITIES", "CDATA","NMTOKEN","NMTOKENS" or "NOTATION". *Action:* Change the document to correct the error and retry.
5045	**XRSN_ATTL_ENUMLIST_CHAR_INVALID** Incorrect character is found in the attribute list declaration enumerated list. *Action:* Change the document to correct the error and retry.
5046	**XRSN_ATTL_DEFVALUE_CHAR_INVALID** Incorrect character is found in attribute list declaration default. Expected white space, "#", or a single or double quote *Action:* Change the document to correct the error and retry.
5047	**XRSN_ATTL_DEF_VALUE_INVALID** Incorrect character is found in attribute list declaration default value. Expected "REQUIRED", "IMPLIED", or "FIXED". *Action:* Change the document to correct the error and retry.
5048	**XRSN_ATTL_QUOTE_MISSING** Incorrect character is found in attribute list declaration default value. Expected single quote, double quote or white space. *Action:* Change the document to correct the error and retry.
5049	**XRSN_ATTL_REF_CHAR_INVALID** An incorrect character was found in entity reference in an attribute list declaration. *Action:* Change the document to correct the error and retry.
5050	**XRSN_ATTL_REF_NAME_INVALID** An incorrect character was found in entity reference in an attribute list declaration. *Action:* Change the document to correct the error and retry.
5051	**XRSN_ATTL_REF_VALUE_INVALID** Incorrect character found in character entity reference in an attribute list declaration. *Action:* Change the document to correct the error and retry.

Reason code value	
7001	**XRSN_OIMA_NOT_INITIALIZED** The OIMA provided is unusable. *Action:* Change the schema and retry.
7002	**XRSN_OIMA_NOT_USABLE** The OIMA provided is unusable because a previous reset failed. *Action:* Change the schema and retry.
7003	**XRSN_OIMA_SMALL** The OIMA provided is too small. *Action:* Change the schema and retry.
7005	**XRSN_OIMA_RESIDUAL_DATA** The OIMA is already initialized. *Action:* Change the schema and retry.
7007	**XRSN_JVM_START_FAILED** The Java Virtual Machine failed to start. *Action:* Change the schema and retry.
7008	**XRSN_JVM_STOP_FAILED** The Java Virtual Machine failed to stop. *Action:* Change the schema and retry.
7009	**XRSN_CTLOPTN_UNSUPPORTED** The operation specified for the control parameter is unsupported. *Action:* Ensure that the control options specified are valid when specified together.
7010	**XRSN_ALTOSR_NOTLOADED** The Alternate OSR code is not loaded. *Action:* Change the schema and retry.
7011	**XRSN_JAVACLASS_NOT_FOUND** Java class not found by the ClassLoader. *Action:* Change the schema and retry.
7019	**XRSN_FUNC_NAME_NULL** The specified function name is null. *Action:* Change the schema and retry.
7021	**XRSN_DLL_OPEN_FAILED** Open for the specified DLL failed. *Action:* Change the schema and retry.

Reason code value	
7023	**XRSN_FUNC_RETRIEVE_FAILED** Retrieve for the specified DLL function failed. *Action:* Change the schema and retry.
7027	**XRSN_JAVA_METHOD_NOT_FOUND** The Java method cannot be found in the class. See the diagnostic area for the method name. *Action:* Change the schema and retry.
7029	**XRSN_JAVA_METHOD_CALL_FAILED** A Java method call failed. *Action:* Change the schema and retry.
7031	**XRSN_DLL_CLOSE_FAILED** Close for the specified DLL failed. *Action:* Change the schema and retry.
7033	**XRSN_JNI_METHOD_FAILED** A JNI method returned with an exception. *Action:* Change the schema and retry.
7035	**XRSN_OBJECT_NOT_CREATED** Failed to create a new Java object. *Action:* Change the schema and retry.
7037	**XRSN_SCHEMA_NOT_LOADED** No schemas have been loaded into the OSR generator. *Action:* Change the schema and retry.
7039	**XRSN_OIMAPTR_NOT_PROVIDED** No OIMA pointer has been specified. *Action:* Change the schema and retry.
7043	**XRSN_GEN_OSR_ASM_FAILED** OSR generation failed in the assemble phase. *Action:* Change the schema and retry.
7045	**XRSN_GEN_OSR_COMP_FAILED** OSR generation failed in the compile phase. *Action:* Change the schema and retry.
7046	**XRSN_GEN_OSR_FAILED** OSR generation failed. *Action:* Change the schema and retry.

Reason code value	
7049	**XRSN_OSR_NOT_VALID** The OSR to load is not valid. *Action:* Change the schema and retry.
7050	**XRSN_OSR_MALLOC_FAILED** The OSR generator could not allocate memory. *Action:* Change the schema and retry.
7051	**XRSN_OSR_MFREE_FAILED** The OSR generator could not free memory. *Action:* Change the schema and retry.
7055	**XRSN_JAVAEXCEPTION_DIAG_FAILED** Could not save the Java exception in the diagnostic area. *Action:* Change the schema and retry.
7057	**XRSN_JAVAEXCEPTION_INCOMPLETE** The Java exception saved in the diagnostic area is incomplete. *Action:* Change the schema and retry.
7059	**XRSN_JAVARSNCODE_NOT_FOUND** Unable to obtain the reason code set by the Java exception. *Action:* Change the schema and retry.
7061	**XRSN_INCORRECT_SCHEMA_URI** The URI specified is incorrect. *Action:* Change the schema and retry.
7063	**XRSN_JAVARSNCODE_UNKNOWN** No specific reason code was set by Java. *Action:* Change the schema and retry.
7065	**XRSN_SCHEMA_URI_NOT_FOUND** The schema identified by the specified URI is not found. *Action:* Change the schema and retry.
7067	**XRSN_SCHEMA_LOAD_FAILED** Unable to load the specified schema. *Action:* Change the schema and retry.
7069	**XRSN_OSR_URI_NOT_FOUND** The OSR identified by the specified URI is not found. *Action:* Change the schema and retry.

Reason code value	
7071	**XRSN_STRINGID_SYSSVC_NULL** The system service parameter specified is null. *Action:* Change the schema and retry.
7079	**XRSN_JAVAERRORMESSAGE_INCOMPLETE** The Java error information saved in the diagnostic area is incomplete. *Action:* Change the schema and retry.
7081	**XRSN_SCHEMA_INCORRECT** The specified schema contains an error that caused an exception. *Action:* Change the schema and retry.
7082	**XRSN_SCHEMA_WARNING** The specified schema contains an error that caused a warning. *Action:* Change the schema and retry.
7083	**XRSN_JAVAERRORMESSAGE_DIAG_FAILED** The Java error information saved in the diagnostic area is not valid. *Action:* Change the schema and retry.
7087	**XRSN_OSR_UNSUPPORTED_FEATURE** An unsupported feature flag was specified. *Action:* Change the schema and retry.
7089	**XRSN_OSR_PARM_NOT_SPECIFIED** No OSR parameter was specified. *Action:* Change the schema and retry.
7091	**XRSN_SCHEMA_PARM_NOT_SPECIFIED** No schema parameter was specified. *Action:* Change the schema and retry.
7093	**XRSN_STRIDTBL_PARM_NOT_SPECIFIED** No stringID table parameter was specified. *Action:* Change the document to correct the error and retry.
7095	**XRSN_JAVAPROPERTY_MALFORMED_URL** A well-formed URL could not be constructed for the specified class. *Action:* Contact your system administrator.
7096	**XRSN_ENTITY_RESOLVER_NOTFOUND** The entity resolver could not be found. *Action:* Change the schema and retry.

Reason code value	
7097	**XRSN_JAVAPROPERTY_CLASS_NOTFOUND** The OSR generator classes could not be found. *Action:* Contact your system administrator.
7099	**XRSN_CLSLOADER_ACCESS_FAILED** The OSR generator classes could not be loaded. *Action:* Contact your system administrator.
7101	**XRSN_CLSLOADER_INSTANTIATION_FAILED** The OSR generator classes could not be instantiated. *Action:* Contact your system administrator.
7103	**XRSN_OSR_NOT_LOADED** No OSRs have been loaded into the OSR generator. *Action:* Change the schema and retry.
7107	**XRSN_JVM_OUT_OF_MEMORY** The Java Virtual Machine is out of memory. *Action:* Contact your system administrator.
7109	**XRSN_JVM_STACK_OVERFLOW** The Java Virtual Machine stack overflow occurs. *Action:* Contact your system administrator.
7111	**XRSN_JVM_INTERNAL_ERROR** Internal error has occurred in the Java Virtual Machine. *Action:* Contact your system administrator.
7113	**XRSN_JVM_UNKNOWN_ERROR** An unknown and serious exception has occurred in the JVM. *Action:* Contact your system administrator.
8000	**XRSN_XML_QUOTEREQUIREDINENTITYVALUE** An entity value must begin with a single or double quote. *Action:* Change the document or schema to correct and retry.
8001	**XRSN_XML_INVCHARINENTITYVALUE** An invalid XML character was found in the literal entity value. *Action:* Change the document or schema to correct and retry.
8002	**XRSN_XML_INVCHARINSYSTEMID** An invalid XML character was found in a system identifier. *Action:* Change the document or schema to correct and retry.

Reason code value	
8003	**XRSN_XML_INVCHARINPUBLICID** An invalid XML character was found in a public identifier. *Action:* Change the document or schema to correct and retry.
8004	**XRSN_XML_INVCHARINDOCTYPEDECL** An invalid XML character was found in a document declaration. *Action:* Change the document or schema to correct and retry.
8005	**XRSN_XML_INVCHARININTERNALSUBSET** An invalid XML character found in the internal subset of the DTD. *Action:* Change the document or schema to correct and retry.
8006	**XRSN_XML_INVCHARINEXTERNALSUBSET** An invalid XML character found in the external subset of the DTD. *Action:* Change the document or schema to correct and retry.
8007	**XRSN_XML_INVCHARINIGNORESECT** An invalid XML character was found in the excluded conditional section. *Action:* Change the document or schema to correct and retry.
8008	**XRSN_XML_QUOTEREQUIREDINSYSTEMID** A system identifier must begin with either a single or double quote. *Action:* Change the document or schema to correct and retry.
8009	**XRSN_XML_SYSTEMIDUNTERMINATED** A system identifier must end with a matching quote. *Action:* Change the document or schema to correct and retry.
8010	**XRSN_XML_QUOTEREQUIREDINPUBLICID** A public identifier must begin with a single or double quote. *Action:* Change the document or schema to correct and retry.
8011	**XRSN_XML_PUBLICIDUNTERMINATED** A public identifier must end with a matching quote. *Action:* Change the document or schema to correct and retry.
8012	**XRSN_XML_PUBIDCHARILLEGAL** A public identifier character is not permitted. *Action:* Change the document or schema to correct and retry.
8013	**XRSN_XML_ENTITYVALUEUNTERMINATED** The literal value for the entity must end with a matching quote. *Action:* Change the document or schema to correct and retry.

Reason code value	
8014	**XRSN_XML_SPACEREQDINDECL** White space is required after DOCTYPE in the document type declaration. *Action:* Change the document or schema to correct and retry.
8015	**XRSN_XML_ROOTELEMENTTYPEREQUIRED** A root element type must appear after DOCTYPE in the document type declaration. *Action:* Change the document or schema to correct and retry.
8016	**XRSN_XML_DOCTYPEDECLUNTERMINATED** A document type declaration for the root element type must end with a ">". *Action:* Change the document or schema to correct and retry.
8017	**XRSN_XML_PEREFERENCEWITHINMARKUP** A parameter entity reference cannot occur within markup in the internal subset of the DTD. *Action:* Change the document or schema to correct and retry.
8018	**XRSN_XML_PEREFINCOMPLETEMARKUP** A parameter entity reference cannot occur within the internal subset of the DTD. *Action:* Change the document or schema to correct and retry.
8019	**XRSN_XML_MARKUPNORECOGNIZEDINDTD** The markup declarations contained or pointed to by the document type declaration must be well-formed. *Action:* Change the document or schema to correct and retry.
8020	**XRSN_XML_XMLSPACEDECLARATIONILLEGAL** The attribute declaration for xml:space must be given an enumerated type whose only possible values are default and preserve. *Action:* Change the document or schema to correct and retry.
8021	**XRSN_XML_SPACEREQDETYPEINEDECL** A space is required before an element type. *Action:* Change the document or schema to correct and retry.
8022	**XRSN_XML_ETYPEREQDINELEMENTDECL** An element type is required in an element declaration. *Action:* Change the document or schema to correct and retry.
8023	**XRSN_XML_SPACEREQDINELEMENTDEC** White space is required after the element type in the element type declaration. *Action:* Change the document or schema to correct and retry.

Reason code value	
8024	**XRSN_XML_CONTENTSPECREQDINEDECL** A constraint is required after the element type in the element type declaration. *Action:* Change the document or schema to correct and retry.
8025	**XRSN_XML_ELEMENTDECLUNTERMINATED** The declaration for an element must end with ">". *Action:* Change the document or schema to correct and retry.
8026	**XRSN_XML_OPENPARENORELEREQDINCHIL** A "(" or an element type is required in the declaration of an element. *Action:* Change the document or schema to correct and retry.
8027	**XRSN_XML_CLOSEDPARENREQDINCHIL** A ")" is required in the declaration. *Action:* Change the document or schema to correct and retry.
8028	**XRSN_XML_ELEMTYPEREQDINMIXEDCON** An element type is required in mixed content. *Action:* Change the document or schema to correct and retry.
8029	**XRSN_XML_CLOSEPARENTREQDINMIXEDCON** A ")" is required in the declaration of an element. *Action:* Change the document or schema to correct and retry.
8030	**XRSN_XML_MIXEDCONTENTUNTERMINATED** The mixed content model must end with ")*" when the types of child elements are constrained. *Action:* Change the document or schema to correct and retry.
8031	**XRSN_XML_SPACEREQDINATTLISTDECL** White space is required after !ATTLIST in an attribute list declaration. *Action:* Change the document or schema to correct and retry.
8032	**XRSN_XML_ELEMTYPEREQDINATTLISTDECL** An element type is required in an attribute list declaration. *Action:* Change the document or schema to correct and retry.
8033	**XRSN_XML_SPACEREQDINATTDEF** White space is required after !ATTLIST in an attribute list declaration. *Action:* Change the document or schema to correct and retry.
8034	**XRSN_XML_ATTRNAMEREQDINATTDEF** The attribute name must be specified in the attribute list declaration for the element. *Action:* Change the document or schema to correct and retry.

Reason code value	
8035	**XRSN_XML_SPACEREQDBATINATTDEF** White space is required before an attribute type in an attribute list declaration. *Action:* Change the document or schema to correct and retry.
8036	**XRSN_XML_ATTTYPEREQDINATTDEF** The attribute type is required in the declaration of the attribute for the element. *Action:* Change the document or schema to correct and retry.
8037	**XRSN_XML_SPACEREQDBDDINATTDEF** White space is required before the default declaration in an attribute list declaration. *Action:* Change the document or schema to correct and retry.
8038	**XRSN_XML_DEFDECLREQDINATTDEF** The attribute default is required in the declaration in an attribute list declaration. *Action:* Change the document or schema to correct and retry.
8039	**XRSN_XML_SPACEREQDANOTINNOTTYPE** White space must follow NOTATION in the attribute declaration. *Action:* Change the document or schema to correct and retry.
8040	**XRSN_XML_OPENPARENREQDINNOTTYPE** The "(" character must follow NOTATION in the attribute declaration. *Action:* Change the document or schema to correct and retry.
8041	**XRSN_XML_NAMEREQDINNOTTYPE** The notation name is required in the notation type list for the attribute declaration. *Action:* Change the document or schema to correct and retry.
8042	**XRSN_XML_NOTTYPEUNTERMINATED** The notation type list must end with a ")" in the attribute declaration. *Action:* Change the document or schema to correct and retry.
8043	**XRSN_XML_NMTOKREQDINENUM** The name token is required in the enumerated type list for the attribute declaration. *Action:* Change the document or schema to correct and retry.
8044	**XRSN_XML_ENUMUNTERMINATED** The enumerated type list must end with ")" in the attribute declaration. *Action:* Change the document or schema to correct and retry.

Reason code value	
8045	**XRSN_XML_SPACEREQDINDEFDECL** White space must appear after FIXED in the attribute declaration. *Action:* Change the document or schema to correct and retry.
8046	**XRSN_XML_INCLUDESECTUNTERMINATED** The included conditional section must end with "". *Action:* Change the document or schema to correct and retry.
8047	**XRSN_XML_IGNORESECTUNTERMINATED** The excluded conditional section must end with "". *Action:* Change the document or schema to correct and retry.
8048	**XRSN_XML_NAMEREQDINPEREF** The entity name must immediately follow the "%" in the parameter entity reference. *Action:* Change the document or schema to correct and retry.
8049	**XRSN_XML_SEMICOLONREQDINPEREF** The parameter entity reference must end with the semicolon delimiter. *Action:* Change the document or schema to correct and retry.
8050	**XRSN_XML_SPACEREQDBENINENTITYDECL** White space is required before the entity name in the entity declaration. *Action:* Change the document or schema to correct and retry.
8051	**XRSN_XML_SPACEREQDBPINPEDECL** White space is required before the percent sign in the parameter entity declaration. *Action:* Change the document or schema to correct and retry.
8052	**XRSN_XML_SPACEREQDBEINPEDECL** White space is required between the "%" and the entity name in the parameter entity declaration. *Action:* Change the document or schema to correct and retry.
8053	**XRSN_XML_ENTITYNAMEREQINEDECL** The name of the entity is required in the entity declaration. *Action:* Change the document or schema to correct and retry.
8054	**XRSN_XML_SPACEREQDAENAMEINEDECL** White space is required between the entity name and the definition in the entity declaration. *Action:* Change the document or schema to correct and retry.

Reason code value	
8055	**XRSN_XML_SPACEREQDBNDATAINUEDECL** White space is required before NDATA in the declaration for the entity. *Action:* Change the document or schema to correct and retry.
8056	**XRSN_XML_SPACEREQDBNNAMEINUEDECL** White space is required between "NDATA" and the notation name in the declaration for the entity. *Action:* Change the document or schema to correct and retry.
8057	**XRSN_XML_NOTATIONNAMEREQDINUEDECL** The notation name is required after NDATA in the declaration for the entity. *Action:* Change the document or schema to correct and retry.
8058	**XRSN_XML_ENTITYDECLUNTERMINATED** The declaration for the entity must end with ">". *Action:* Change the document or schema to correct and retry.
8059	**XRSN_XML_EXTERNALIDREQD** The external entity declaration must begin with either SYSTEM or PUBLIC. *Action:* Change the document or schema to correct and retry.
8060	**XRSN_XML_SPACEREQDBPLINEXTERNALID** White space is required between PUBLIC and the public identifier. *Action:* Change the document or schema to correct and retry.
8061	**XRSN_XML_SPACEREQDAPLINEXTERNALID** White space is required between the public identifier and the system identifier. *Action:* Change the document or schema to correct and retry.
8062	**XRSN_XML_SPACEREQDBSLINEXTERNALID** White space is required between SYSTEM and the system identifier. *Action:* Change the document or schema to correct and retry.
8063	**XRSN_XML_URIFRAGINSYSTEMID** The fragment identifier should not be specified as part of the system identifier. *Action:* Change the document or schema to correct and retry.
8064	**XRSN_XML_SPACEREQDBNNINNOTATIONDECL** White space is required before the notation name in the notation declaration. *Action:* Change the document or schema to correct and retry.

Reason code value	
8065	**XRSN_XML_NOTATIONNAMEREQDINNOTDECL** The name of the notation is required in the notation declaration. *Action:* Change the document or schema to correct and retry.
8066	**XRSN_XML_SPACEREQDANNINNOTATIONDECL** White space is required after the notation name in the notation declaration. *Action:* Change the document or schema to correct and retry.
8067	**XRSN_XML_NOTATIONDECLUNTERMINATED** The declaration for the notation must end with a ">". *Action:* Change the document or schema to correct and retry.
8068	**XRSN_XML_UNDECLAREDELEMINCONTSPEC** The content model of the element refers to the undeclared element. *Action:* Change the document or schema to correct and retry.
8069	**XRSN_XML_DUPLICATEATTDEF** There is a duplicate attribute definition found. *Action:* Change the document or schema to correct and retry.
8070	**XRSN_XML_ROOTELEMTMUSTMATCHDOCTDECL** The root element type must match the document type declaration. *Action:* Change the document or schema to correct and retry.
8071	**XRSN_XML_IMPROPERDECLNESTING** The replacement text of a parameter entity must include properly nested declarations. *Action:* Change the document or schema to correct and retry.
8072	**XRSN_XML_WSINELEMCONTENTWHENSA** White space must not occur between elements declared in an external parsed entity with element content in a standalone document. *Action:* Change the document or schema to correct and retry.
8073	**XRSN_XML_REFTOEXTDECLAREDENTWHENSA** The reference to an entity declared in an external parsed entity is not permitted in a standalone document. *Action:* Change the document or schema to correct and retry.
8074	**XRSN_XML_EXTENTITYNOTPERMITED** The reference to an external entity is not permitted in a standalone document. *Action:* Change the document or schema to correct and retry.

Reason code value	
8075	**XRSN_XML_ATTVALCHANGEDDURNORMWHENSA** The value of an attribute must not be changed by normalization in a standalone document. *Action:* Change the document or schema to correct and retry.
8076	**XRSN_XML_DEFATTNOTSPECIFIED** An attribute has a default value and must be specified in a standalone document. *Action:* Change the document or schema to correct and retry.
8077	**XRSN_XML_CONTENTINCOMPLETE** The content of an element type is incomplete. *Action:* Change the document or schema to correct and retry.
8078	**XRSN_XML_CONTENTINVALID** The content is invalid. *Action:* Change the document or schema to correct and retry.
8079	**XRSN_XML_ELEMENTNOTDECLARED** An element must be declared. *Action:* Change the document or schema to correct and retry.
8080	**XRSN_XML_ATTRIBUTENOTDECLARED** An attribute must be declared. *Action:* Change the document or schema to correct and retry.
8081	**XRSN_XML_ELEMENTALREADYDECLARED** An element type must not be declared more than once. *Action:* Change the document or schema to correct and retry.
8082	**XRSN_XML_IMPROPERGROUPNESTING** The replacement text of a parameter entity must include properly nested pairs of parentheses. *Action:* Change the document or schema to correct and retry.
8083	**XRSN_XML_DUPTYPEINMIXEDCONTENT** A duplicate type found in a mixed content declaration. *Action:* Change the document or schema to correct and retry.
8084	**XRSN_XML_NOTATIONONEMPTYELEMENT** For compatibility, an attribute of type NOTATION must not be declared on an element declared EMPTY. *Action:* Change the document or schema to correct and retry.

Reason code value

8085 **XRSN_XML_ENTITIESINVALID**

Attribute value of type ENTITIES must be the name of one or more unparsed entities.

Action: Change the document or schema to correct and retry.

8086 **XRSN_XML_ENTITYINVALID**

An attribute value of type ENTITY must be the name of an unparsed entity.

Action: Change the document or schema to correct and retry.

8087 **XRSN_XML_IDDEFTYPEINVALID**

An ID attribute must have a declared default of #IMPLIED or #REQUIRED.

Action: Change the document or schema to correct and retry.

8088 **XRSN_XML_IDINVALID**

An attribute value of type ID must be a name.

Action: Change the document or schema to correct and retry.

8089 **XRSN_XML_IDNOTUNIQUE**

An attribute value of type ID must be unique within the document.

Action: Change the document or schema to correct and retry.

8090 **XRSN_XML_IDREFINVALID**

An attribute value of type IDREF must be a name.

Action: Change the document or schema to correct and retry.

8091 **XRSN_XML_IDREFSINVALID**

An attribute value of type IDREFS must be one or more names.

Action: Change the document or schema to correct and retry.

8092 **XRSN_XML_ATTVALUENOTINLIST**

An attribute value is not in the list.

Action: Change the document or schema to correct and retry.

8093 **XRSN_XML_NMTOKENINVALID**

An attribute value of type NMTOKENS must be a name token.

Action: Change the document or schema to correct and retry.

8094 **XRSN_XML_NMTOKENSINVALID**

An attribute value for type NMTOKENS must be one or more name tokens.

Action: Change the document or schema to correct and retry.

Reason code value	
8095	**XRSN_XML_ELEMWITHIDREQD** An element with an ID is required. *Action:* Change the document or schema to correct and retry.
8096	**XRSN_XML_MORETHANONEIDATTR** A second attribute of type ID is not permitted. *Action:* Change the document or schema to correct and retry.
8097	**XRSN_XML_MORETHANONENOTATTR** A second attribute of type NOTATION is not permitted. *Action:* Change the document or schema to correct and retry.
8098	**XRSN_XML_DUPTOKENINLIST** The enumerated type list must not contain duplicate tokens. *Action:* Change the document or schema to correct and retry.
8099	**XRSN_XML_FIXATTVALUEINVALID** A FIXED attribute value is invalid. *Action:* Change the document or schema to correct and retry.
8100	**XRSN_XML_REQDATTNOTSPECIFIED** An attribute is required and must be specific for the element type. *Action:* Change the document or schema to correct and retry.
8101	**XRSN_XML_ATTDEFINVALID** The default value must meet the lexical type constraints declared for the attribute. *Action:* Change the document or schema to correct and retry.
8102	**XRSN_XML_IMPROPERCONDSECTNESTING** The replacement text of the parameter entity must include properly nested conditional sections. *Action:* Change the document or schema to correct and retry.
8103	**XRSN_XML_NOTATIONNOTDECLFORNOTTATT** The notation must be declared when referenced in the notation type list for the attribute. *Action:* Change the document or schema to correct and retry.
8104	**XRSN_XML_NOTATIONNOTDECLFORUPEDECL** The notation must be declared when referenced in the unparsed entity declaration. *Action:* Change the document or schema to correct and retry.

Reason code value	
8105	**XRSN_XML_UNIQUENOTNAME** Only one notation declaration can declare a given name. *Action:* Change the document or schema to correct and retry.
8106	**XRSN_XML_REFTOEXTENTITY** The external entity reference is not permitted in an attribute value. *Action:* Change the document or schema to correct and retry.
8107	**XRSN_XML_PENOTDECLARED** The parameter entity was referenced but not declared. *Action:* Change the document or schema to correct and retry.
8108	**XRSN_XML_REFTOUNPENTITY** The unparsed reference is not permitted. *Action:* Change the document or schema to correct and retry.
8109	**XRSN_XML_RECURSIVEREFERENCE** A recursive reference was found. *Action:* Change the document or schema to correct and retry.
8110	**XRSN_XML_RECURSIVEPEREFERENCE** A recursive PE reference was found. *Action:* Change the document or schema to correct and retry.
8111	**XRSN_XML_ENCODINGNOTSUPPORTED** The encoding is not supported in the entity. *Action:* Change the document or schema to correct and retry.
8112	**XRSN_XML_ENCODINGREQD** A parsed entity not encoded in either UTF-8 or UTF-16 must contain an encoding declaration. *Action:* Change the document or schema to correct and retry.
8200	**XRSN_IMP_UNABLETOCONVERTCHAR** Unable to convert an out of range unicode character. *Action:* Change the document or schema to correct and retry.
8201	**XRSN_IMP_INSUFFINPUTTODECCHAR** There is insufficient input to decode the character. *Action:* Change the document or schema to correct and retry.
8202	**XRSN_IMP_MISSING2NDHALFOFPAIR** A surrogate pair is missing its second half for a unicode character. *Action:* Change the document or schema to correct and retry.

Reason code value	
8203	**XRSN_IMP_INVAL2NDHALFOFPAIR** An invalid second half of a surrogate pair for a unicode character was found. *Action:* Change the document or schema to correct and retry.
8204	**XRSN_IMP_INVAL1STHALFOFPAIR** An invalid first half of a surrogate pair for a unicode character was found. *Action:* Change the document or schema to correct and retry.
8205	**XRSN_IMP_BOMREQD** A byte order mark is required. *Action:* Change the document or schema to correct and retry.
8206	**XRSN_IMP_INVUTF8SURENCODING** An invalid UTF-8 surrogate encoding found. *Action:* Change the document or schema to correct and retry.
8207	**XRSN_IMP_PARTIALMPCHARSEQ** A partial multipart character sequence found. *Action:* Change the document or schema to correct and retry.
8208	**XRSN_IMP_INCONSISTENTENC** An encoding name and byte stream contents are inconsistent. *Action:* Change the document or schema to correct and retry.
8209	**XRSN_IMP_INVUTF8CHARENC** An invalid UTF-8 character encoding was found. *Action:* Change the document or schema to correct and retry.
8210	**XRSN_IMP_RUNTIMEIOERROR** A runtime IO error has occurred. *Action:* Change the document or schema to correct and retry.
8212	**XRSN_MULTIFRAGMENT_NOT_ALLOWED** Multiple elements values are not allowed in the document fragment for validation in fragment parsing. *Action:* Change the document fragment with a single element and retry.
8400	**XRSN_DEM_ROOTELEMENTREQD** The root element is required in a well-formed document. *Action:* Change the document or schema to correct and retry.

Reason code value	
8401	**XRSN_DEM_INVCHARINCDSECT** An invalid XML character was found in the CDATA section of the document. *Action:* Change the document or schema to correct and retry.
8402	**XRSN_DEM_INVCHARINCONTENT** An invalid XML character was found in the element content of the document. *Action:* Change the document or schema to correct and retry.
8403	**XRSN_DEM_INVCHARINMISC** An invalid XML character was found in the markup after the end of the element content. *Action:* Change the document or schema to correct and retry.
8404	**XRSN_DEM_INVCHARINPROLOG** An invalid XML character was found in the prolog of a document. *Action:* Change the document or schema to correct and retry.
8405	**XRSN_DEM_CDENDINCONTENT** The character sequence must not appear in content unless used to mark the end of a CDATA section. *Action:* Change the document or schema to correct and retry.
8406	**XRSN_DEM_CDSECTUNTERMINATED** The CDATA section must end with . *Action:* Change the document or schema to correct and retry.
8407	**XRSN_DEM_EQREQDINXMLDECL** The equal character must follow the keyword in the XML declaration. *Action:* Change the document or schema to correct and retry.
8408	**XRSN_DEM_QUOTEREQDINXMLDECL** This value in the XML declaration must be a quoted string. *Action:* Change the document or schema to correct and retry.
8409	**XRSN_DEM_XMLDECLUNTERMINATED** The XML declaration must end with ?>. *Action:* Change the document or schema to correct and retry.
8410	**XRSN_DEM_VERSIONINFOREQD** The version is required in the XML declaration. *Action:* Change the document or schema to correct and retry.

Reason code value	
8411	**XRSN_DEM_MARKUPNOTRECINPROLOG** The markup in the document preceding the root element must be well-formed. *Action:* Change the document or schema to correct and retry.
8412	**XRSN_DEM_MARKUPNORECINMISC** The markup in the document following the root element must be well-formed. *Action:* Change the document or schema to correct and retry.
8413	**XRSN_DEM_SDDECLINVALID** The standalone document declaration must be yes or no. *Action:* Change the document or schema to correct and retry.
8414	**XRSN_DEM_ETAGREQD** End-tag is required. *Action:* Change the document or schema to correct and retry.
8415	**XRSN_DEM_ELEMUNTERMINATED** The element must be followed by either attribute specifications, > or />. *Action:* Change the document or schema to correct and retry.
8416	**XRSN_DEM_EQREQDINATTR** The attribute name must be followed by the = character. *Action:* Change the document or schema to correct and retry.
8417	**XRSN_DEM_ATTRNOTUNQ** The attribute was already specified for the element. *Action:* Change the document or schema to correct and retry.
8418	**XRSN_DEM_ETAGUNTERM** The end-tag for the element must end with a > delimiter. *Action:* Change the document or schema to correct and retry.
8419	**XRSN_DEM_MARKUPNORECINCONT** The content of elements must consist of well-formed character data or markup. *Action:* Change the document or schema to correct and retry.
8420	**XRSN_DEM_ELEMENTMISMATCH** The element must start and end within the same entity. *Action:* Change the document or schema to correct and retry.

Reason code value	
8421	**XRSN_DEM_INVALCHARINATTRVAL** An invalid XML character was found in the attribute value. *Action:* Change the document or schema to correct and retry.
8422	**XRSN_DEM_INVALCHARINCOMM** An invalid XML character was found in the comment. *Action:* Change the document or schema to correct and retry.
8423	**XRSN_DEM_INVALCHARINPI** An invalid XML character was found in the processing instruction. *Action:* Change the document or schema to correct and retry.
8424	**XRSN_DEM_QUOTEREQDINATTRVAL** The value of an attribute must begin with either a single or double quote character. *Action:* Change the document or schema to correct and retry.
8425	**XRSN_DEM_LESSTHANINATTRVAL** The value of the attribute must not contain the < character. *Action:* Change the document or schema to correct and retry.
8426	**XRSN_DEM_ATTRVALUNTERM** The attribute value must end with the matching quote character. *Action:* Change the document or schema to correct and retry.
8427	**XRSN_DEM_INVALCOMMSTART** The comment must begin with a comment start sequence. *Action:* Change the document or schema to correct and retry.
8428	**XRSN_DEM_DASHDASHINCOMM** A double hyphen is not allowed in a comment. *Action:* Change the document or schema to correct and retry.
8429	**XRSN_DEM_COMMENTUNTERM** The comment must end with a comment ending sequence. *Action:* Change the document or schema to correct and retry.
8430	**XRSN_DEM_PITARGETREQD** The processing instruction must begin with the name of the target. *Action:* Change the document or schema to correct and retry.
8431	**XRSN_DEM_SPACEREQDINPI** A white space character is required between the processing instruction target and the data. *Action:* Change the document or schema to correct and retry.

Reason code value	
8432	**XRSN_DEM_PIUNTERMINATED** The processing instruction must end with ?>. *Action:* Change the document or schema to correct and retry.
8433	**XRSN_DEM_RESERVEDPITARGET** The processing instruction target matching [xX][mM][lL] is not allowed. *Action:* Change the document or schema to correct and retry.
8434	**XRSN_DEM_VERNOTSUPPORTED** The XML version specified is not supported. *Action:* Change the document or schema to correct and retry.
8435	**XRSN_DEM_DIGREQDINCHARREF** A decimal representation must immediately follow the &# in the character reference. *Action:* Change the document or schema to correct and retry.
8436	**XRSN_DEM_HEXREQDINCHARREF** A hexadecimal representation must immediately follow the &#x in the character reference. *Action:* Change the document or schema to correct and retry.
8437	**XRSN_DEM_SEMICOLONREQDINCHARREF** The character reference must end with a semicolon delimiter. *Action:* Change the document or schema to correct and retry.
8438	**XRSN_DEM_INVCHARREF** The character reference contains an invalid character. *Action:* Change the document or schema to correct and retry.
8439	**XRSN_DEM_NAMEREQDINREF** The entity name must immediately follow the & in the entity reference. *Action:* Change the document or schema to correct and retry.
8440	**XRSN_DEM_SEMICOLONREQDINREF** The reference to the entity must end with a semicolon delimiter. *Action:* Change the document or schema to correct and retry.
8441	**XRSN_DEM_EQREQDINTDECL** The = character is required in the text declaration. *Action:* Change the document or schema to correct and retry.
8442	**XRSN_DEM_QUOTEREQDINTDECL** The value in the text declaration must be a quoted string. *Action:* Change the document or schema to correct and retry.

Reason code value	
8443	**XRSN_DEM_SPACEREQDINTDECL** White space is required between the version and the encoding declaration. *Action:* Change the document or schema to correct and retry.
8444	**XRSN_DEM_TEXTDECLUNTERM** The text declaration must end with ?>. *Action:* Change the document or schema to correct and retry.
8445	**XRSN_DEM_ENCDECLREQD** The encoding is required in the text declaration. *Action:* Change the document or schema to correct and retry.
8446	**XRSN_DEM_ENCDECLINV** The encoding name is invalid. *Action:* Change the document or schema to correct and retry.
8447	**XRSN_DEM_ENTNOTDECL** A general entity was referenced but not declared. *Action:* Change the document or schema to correct and retry.
8448	**XRSN_DEM_COLONINNAME** Namespaces disallow a colon character except in element types or attribute names. *Action:* Change the document or schema to correct and retry.
8449	**XRSN_DEM_TWOCOLONSQN** Namespaces allows only one colon character in element types or attribute names. *Action:* Change the document or schema to correct and retry.
8450	**XRSN_DEM_PREFDECL** The namespace prefix was not declared. *Action:* Change the document or schema to correct and retry.
8451	**XRSN_DEM_PREFLEGAL** The namespace name for prefix xml is not bound to a legal namespace name. *Action:* Change the document or schema to correct and retry.
8452	**XRSN_DEM_NSNAMEEMPTY** The namespace name declared for the prefix may not be empty. *Action:* Change the document or schema to correct and retry.

Reason code value	
8453	**XRSN_DEM_NSRSRD** The namespace prefix is bound to the reserved namespace name. *Action:* Change the document or schema to correct and retry.
8454	**XRSN_DEM_NSPREFRSRD** The namespace prefix "xmlns" must not be declared. *Action:* Change the document or schema to correct and retry.
8500	**XRSN_XDBX_DOCID_INCORRECT** The document identifier for the XDBX stream must be "#xCA #x3B". *Action:* Change the document to correct the error and retry.
8501	**XRSN_XDBX_HDRLEN_INCORRECT** The length of the XDBX document header is a one byte value. This value does not including the magic number or the length byte itself. The value must be at least "#x5" for the XDBX major version 1. *Action:* Change the document to correct the error and retry.
8502	**XRSN_XDBX_VERSION_NOT_SUPPORTED** This version of the XDBX document encoder is not supported. *Action:* Change the document to correct the error and retry.
8503	**XRSN_XDBX_STRIDS_NOT_USED** The stringID encoding flag is missing from the header of the XDBX stream. *Action:* Change the document to correct the error and retry.
8504	**XRSN_XDBX_STRID_NOT_FOUND** An attempt was made to resolve a stringID that has not been specified. *Action:* Change the document to correct the error and retry.
8505	**XRSN_XDBX_STREAM_INCORRECT** One or more bytes from the XDBX input stream are incorrect. *Action:* Change the document to correct the error and retry.
8506	**XRSN_XDBX_TAG_UNEXPECTED** The current tag in the XDBX stream is not expected. *Action:* Change the document to correct the error and retry.
8507	**XRSN_XDBX_SEQ_UNSUPPORTED** Sequences of XDBX items are not supported. *Action:* Change the document to correct the error and retry.
8508	**XRSN_XDBX_STRID_INCORRECT** The value of the StringID is not a legitimate positive number. *Action:* Change the document to correct the error and retry.

Reason code value	
8509	**XRSN_XDBX_STANDALONE_INCORRECT** The standalone value is incorrect. The only recognized values are 0 (FALSE) or 1 (TRUE). *Action:* Change the document to correct the error and retry.
8510	**XRSN_XDBX_MISSING_ROOT_ELEMENT** The XDBX stream requires at least one element and none were found. *Action:* Change the document to correct the error and retry.
8511	**XRSN_XDBX_DUPLICATE_STRID** The StringID value is duplicate of one of the previous ones. *Action:* Change the document to correct the error and retry.
8600	**XRSN_VME_INVATTVALUE** The attribute value is not valid with respect to its type. *Action:* Change the document or schema to correct and retry.
8601	**XRSN_VME_INVATTVALUEFORFIXED** The attribute value is not valid with respect to its fixed value constraint. *Action:* Change the document or schema to correct and retry.
8602	**XRSN_VME_CONTENTFOREMPTYELEM** The element may not contain any character data or child elements because the element type is EMPTY. *Action:* Change the document or schema to correct and retry.
8603	**XRSN_VME_NONWSCHARINELEMONLYCONT** The element cannot have non-white space character data because the type's content type is element-only. *Action:* Change the document or schema to correct and retry.
8604	**XRSN_VME_EXPELEMNOMATCH** An expected element match was not found. *Action:* Change the document or schema to correct and retry.
8605	**XRSN_VME_REQDELEMMISSING** The required element or one of its substitutions is required. *Action:* Change the document or schema to correct and retry.
8606	**XRSN_VME_STRICTWCREQTDECL** The matching wildcard is strict, but no declaration can be found for the element. *Action:* Change the document or schema to correct and retry.

Reason code value	
8607	**XRSN_VME_EXPECTENDTAG** An end tag is expected. Invalid content is found. No child element is expected at this point. *Action:* Change the document or schema to correct and retry.
8608	**XRSN_VME_ELEMNOTINCHOICE** An unexpected element was found. The element was not one of the choices. *Action:* Change the document or schema to correct and retry.
8609	**XRSN_VME_ELEMDUP** A duplicate element or one of its substitutions was found. *Action:* Change the document or schema to correct and retry.
8610	**XRSN_VME_EMPTYTABINCOMPCONT** An empty element tag is not expected. The content of the element is not complete. *Action:* Change the document or schema to correct and retry.
8611	**XRSN_VME_UNEXPECTEDENDELEM** An unexpected end element event is found. The content of the element is incomplete. *Action:* Change the document or schema to correct and retry.
8612	**XRSN_VME_UNDECLATT** The attribute found is not allowed to appear in the element. *Action:* Change the document or schema to correct and retry.
8613	**XRSN_VME_REQDATTMISSING** The attribute must appear on the element. *Action:* Change the document or schema to correct and retry.
8614	**XRSN_VME_MULTIWILDIDS** ID values must be unique. *Action:* Change the document or schema to correct and retry.
8615	**XRSN_VME_WILDIDFORBID** The attribute is a wildcard ID. But there is already an attribute derived from the ID among the attribute uses. *Action:* Change the document or schema to correct and retry.
8616	**XRSN_VME_NONNILLELEM** Attribute "xsi:nil" must not appear on the element, because the nillable property is false. *Action:* Change the document or schema to correct and retry.

Reason code value	
8617	**XRSN_VME_NILFORBIDWFIXEDVC** There must be no fixed value constraint for the element because "xsi:nil" is specified. *Action:* Change the document or schema to correct and retry.
8618	**XRSN_VME_XSITVALINV** The attribute value "xsi:type" of the element is not a valid QName. *Action:* Change the document or schema to correct and retry.
8619	**XRSN_VME_XSITVALDOESNOTEXIST** The value cannot be resolved to a type definition for the element. *Action:* Change the document or schema to correct and retry.
8620	**XRSN_VME_XSITYPEVALNOTALLOWED** The type is not validly derived from the type definition of the element. *Action:* Change the document or schema to correct and retry.
8621	**XRSN_VME_VCINVFORCURTYPE** The value constraint of the element is not a valid default value for the type. *Action:* Change the document or schema to correct and retry.
8622	**XRSN_VME_FIXEDVCFAILURE** The value does not match the fixed value constraint value for the element. *Action:* Change the document or schema to correct and retry.
8623	**XRSN_VME_IDREFMISSINGID** There is no ID/IDREF binding for IDREF. *Action:* Change the document or schema to correct and retry.
8624	**XRSN_VME_ELEMHASABSTYPE** The type definition cannot be abstract for the element. *Action:* Change the document or schema to correct and retry.
8625	**XRSN_VME_INVSIMPLECONT** Invalid value of element. *Action:* Change the document or schema to correct and retry.
8626	**XRSN_VME_DUPKEY** A duplicate key value was declared for an identity constraint. *Action:* Change the document or schema to correct and retry.
8627	**XRSN_VME_DUPUNIQUE** A duplicate unique value was declared for an identity constraint. *Action:* Change the document or schema to correct and retry.

Reason code value	
8628	**XRSN_VME_FIELDMULTMATCH** A field matches more than one value within the scope of its selector. The fields must match unique values. *Action:* Change the document or schema to correct and retry.
8629	**XRSN_VME_KEYNOTENOUGHVALS** Not enough values were specified for a key identity constraint. *Action:* Change the document or schema to correct and retry.
8630	**XRSN_VME_IDCKEYREFMISSINGKEY** A keyref is missing a corresponding key. *Action:* Change the document or schema to correct and retry.
8631	**XRSN_VME_ABSELEMERROR** The abstract element cannot be used to validate the element content. *Action:* Change the document or schema to correct and retry.
8632	**XRSN_VME_UNEXPECTEDROOT** The root element is not defined in the schema. *Action:* Change the document or schema to correct and retry.
8800	**XRSN_DVE_SIMPLETYPEINVVAL** Simple type is invalid. *Action:* Change the document or schema to correct and retry.
8801	**XRSN_DVE_IDMULTVAL** There are multiple occurrences of the ID value. *Action:* Change the document or schema to correct and retry.
8802	**XRSN_DVE_FACETLENVAL** The value is not facet-valid with respect to the length for this type. *Action:* Change the document or schema to correct and retry.
8803	**XRSN_DVE_FACETMAXEXCVAL** The value is not facet-valid with respect to maxExclusive for this type. *Action:* Change the document or schema to correct and retry.
8804	**XRSN_DVE_FACETMAXINCVAL** The value is not facet-valid with respect to maxInclusive for this type. *Action:* Change the document or schema to correct and retry.
8805	**XRSN_DVE_FACETMAXLENVAL** The value is not facet-valid with respect to maxLength for this type. *Action:* Change the document or schema to correct and retry.

Reason code value	
8806	**XRSN_DVE_FACETMINEXCVAL** The value is not facet-valid with respect to minExclusive for this type. *Action:* Change the document or schema to correct and retry.
8807	**XRSN_DVE_FACETMININCVAL** The value is not facet-valid with respect to minInclusive for this type. *Action:* Change the document or schema to correct and retry.
8808	**XRSN_DVE_FACETMINLENVAL** The value is not facet-valid with respect to minLength for this type. *Action:* Change the document or schema to correct and retry.
8809	**XRSN_DVE_FACETPATTERNVAL** The value is not facet-valid with respect to the pattern for this type. *Action:* Change the document or schema to correct and retry.
8810	**XRSN_DVE_FACETTOTDIGVAL** The value has a mismatch in total number of digits for the type. *Action:* Change the document or schema to correct and retry.
8811	**XRSN_DVE_FACETFRACTDIGVAL** The value has a mismatch in fraction digits for this type. *Action:* Change the document or schema to correct and retry.
8812	**XRSN_DVE_FACETENUMVAL** The value is not facet-valid with respect to the enumeration for this type. It must be a value from the enumeration. *Action:* Change the document or schema to correct and retry.
8900	**XRSN_FRAG_FRAGPATH_ERROR** For each element in the fragment path, a forward slash must be included following by a valid Qname. *Action:* Change the document or schema to correct and retry.
8901	**XRSN_FRAG_INFO_NOTFOUND** The generated OSR must have fragment parsing information inorder to perform a fragment parse. *Action:* Change the document or schema to correct and retry.
8902	**XRSN_FRAG_SLASH_AFTER_ATTR** The attribute name must be the last thing in the fragment path. *Action:* Change the document or schema to correct and retry.

Reason code value	
8903	**XRSN_FRAG_ELEMATTR_NOTFOUND** The element or the attribute name in the fragment path cannot be found in the OSR. *Action:* Change the document or schema to correct and retry.
8904	**XRSN_FRAG_INVALID_TYPE** The declared type in the OSR is invalid. *Action:* Change the document or schema to correct and retry.
8905	**XRSN_FRAG_ATTR_INVALID** During the validation of the attribute value with the = OSR shows the attribute value is invalid. *Action:* Change the document or schema to correct and retry.
8906	**XRSN_FRAG_ATTR_ERROR** Error parsing an attribute fragment. *Action:* Change the document or schema to correct and retry.
8907	**XRSN_FRAG_ATTR_QUOTE_MISSING** A matching single/double quotes are required for the attribute value passed in as the fragment. *Action:* Change the document or schema to correct and retry.
8908	**XRSN_FRAG_ATTR_UNTERMINATED** A matching single/double quotes are required for the attribute value passed in as the fragment. *Action:* Change the document or schema to correct and retry.
8909	**XRSN_FRAG_ATTR_QUOTE_INCORRECT** The attribute value must be contained within a matching single/double quote, and no characters are allowed after the ending quote except whitespaces. *Action:* Change the document or schema to correct and retry.
8910	**XRSN_CTL_RESET_REQUIRED** A Control Reset call is required. *Action:* Prior parse has detected, issue control reset the parser and retry.

Appendix C. xsdosrg command reference

Name

xsdosrg - generate an optimized schema representation (OSR) file

Synopsis

xsdosrg *[-v] [-o output_file] [-l list_file] | (input_file [input_file ...])*

Note: The l option signifies a lower case L, not an upper case I. The option signifies lower case O, not zero.

Description

A z/OS UNIX shell command that creates an optimized schema representation (OSR) from one or more schemas which can be used by the z/OS XML System Services validating parser.

Options

xsdosrg accepts the following command line switches:

-v This option produces verbose output during the generation of the OSR. This is for problem determination purposes only.

-o This option identifies the name of the output file that will contain the generated OSR.

-l This option identifies the list of file names containing the text schemas to process.

Operands

xsdosrg contains the following operands:

input_file
The name of the file containing the text version of an XML schema. At least one input file must be specified, either with this operand, or through the file list operand.

list_file A list of schema names in text form that will be used to create the optimized schema representation. The text in this file must be in the current local codepage so that the command can open each file in the list.

output_file
The output_file operand is the name of the file that will contain the optimized schema representation. This file name defaults to out.osr if no name is specified.

Example

```
xsdosrg -o myschema.osr myschema.xsd
```

Environment variables

See “Setting up the environment” on page 15 for information on setting and using environment variables.

Usage notes

One or more schemas may be processed by the **xsdosrg** command into a single optimized schema representation. Multiple schema names may be specified either directly on the command line or using the file list operand with the -l option. Use either the input file operand or the list option to specify a list of schemas to process. Do not use both methods on the same command invocation.

This command provides a simplified interface to the OSR generation utility. See “gxluGenOSR — generate an Optimized Schema Representation (OSR)” on page 98, which allows greater control over the behavior of the generation process and the characteristics of the generated OSR.

The codepage of the text contained in the list file for the -l option is managed in the same way as any other z/OS UNIX System Services command (for example, **cp**). The localization variables above and file tags may be used to set the proper code page so that file names can be handled properly.

Exit values

The following list contains the exit values generated by this command:

0	Success
4	No schema specified
16	OSR creation failed

Related information

gxluGenOSR is a C routine that also invokes the OSR generator. It provides greater control over the behavior of the generation process and the characteristics of the generated OSR. See “gxluGenOSR — generate an Optimized Schema Representation (OSR)” on page 98 for more information.

gxlOSRGenerator is a Java method that can be used to invoke the OSR generator. Information on this method can be found in the Java API.

Appendix D. C/C++ header files and assembler macros

The z/OS XML System Services API includes several sets of structures, variables and constants that the caller uses to provide input to and receive output from the assorted processing services of the API. These definitions are contained in parallel sets of C/C++ header files and assembler macros. The header files are named gxlh*.h, and are found in the `/usr/include` directory. The assembler macros are named GXLY* and are installed in SYS1.MACLIB.

The names of the C/C++ and assembler macros are similar. For example, the output buffer record mapping is contained in `/usr/include/gxlhxeh.h`, while the assembler version of the same mapping is in SYS1.MACLIB(GXLYXEH). In addition to the parallel nature of these headers and macros, the C/C+ headers come in regular Language Environment run-time and Metal C versions. Both versions have the same file names, but the Language Environment run-time versions are in `/usr/include`, while the Metal C versions are in `/usr/include/metal`. See "z/OS XML XL C/C++ API" on page 54 for more details about these differences.

All of the core parser services have C/C++ interfaces (both Language Environment C and Metal C) and assembler interfaces. In addition, there are a set of utility services to generate Optimized Schema Representations (OSRs) from text schemas. These utility services are implemented in Language Environment C/C++ and Java. As a result, there are Language Environment C/C++ headers that have no corresponding assembler macro or Metal C version.

These are the header files and assembler macros of the z/OS XML processing API. The header file names are listed first, followed by the assembler macro names in parentheses (if there is a corresponding macro).

gxlhxml.h - main z/OS XML header file

This is the main z/OS XML C/C++ header file that a caller should include in order to use the z/OS XML C/C++ API. It contains prototypes for all of the API entry points, as well as include statements for all of the other header files that are required for the API. The Metal C version of this header also includes logic to call either the 31 or 64 bit version of the requested API, depending on the addressing mode of the caller.

There is no corresponding assembler version of this header file.

gxlhxeh.h (GXLYXEH) - mapping of the output buffer record

This mapping describes the form of the parsed data stream returned from the z/OS XML parser. It contains the following:

- A structure describing the fixed portion of a record in the data stream. This includes the record type and assorted flags describing the characteristics of the record.
- A structure to map the length value pairs (if there are any) that make up the variable portion of the record.

- A structure describing the format of string identifiers (StringIDs) used to represent the strings associated with a record when the StringIDs feature is enabled.
- Structures to map the special records that represent buffer information (data stream metadata), error information, and auxiliary information.

The items defined in this mapping provide a complete interface for the caller to make use of the parsed data stream returned from a parse request. See Chapter 4, “Parsing XML documents,” on page 11for more a more detailed explanation of the z/OS XML parsed data stream.

gxlhxec.h (GXLYXEC) - constants definitions

This header and assembler macro contain constant values that are a key part of the z/OS XML API. They include the following:

- Record/token types. These identify the semantic meaning of a record in the parsed data stream.
- Feature flags. These are the z/OS XML parser features that the caller enables when making an initialization or control request.
- Minimum work area sizes for the z/OS XML parser and query XML declaration services. There are unique minimum work area sizes for the z/OS XML parser, depending on whether or not validation is required.
- The minimum output buffer size.
- The allowable option flag values for the control function service.
- Assorted OSR generator constants.
- CCSID constants for all of the encodings that z/OS XML supports.
- Type identifiers for the data contained in source offset information records.

This is the header (macro) that contains all of the well known and required values for the z/OS XML API.

gxlhqxd.h (GXLYQXD) - mapping of the output from the query XML declaration service

This header (macro) contains the structure that describes the information returned from the Query XML Declaration (QXD) service. It also contains constants that enumerate the allowable values for certain fields of the structure. The types of data returned in this area include the following:

- The type of encoding that the service was able to auto-detect. This is not a CCSID, but an indication as to whether the document is in UCS, UTF, or EBCDIC form. It also gives an indication of whether the document is big-endian or little-endian for certain encoding types.
- The CCSID of the document that the service was able to auto-detect. This value is suitable to pass to the z/OS XML parser initialization service to let the z/OS XML parser know the encoding of the document.

  Note: The QXD service is capable of detecting CCSIDs that are not supported by the z/OS XML parser.
- The version and release number from the "version" keyword value in the XML declaration.
- The CCSID from the "encoding" keyword value in the XML declaration. It may be the case that the detected encoding does not match the CCSID from the XML declaration. This could happen if the document has been transcoded from the

original encoding to the detected encoding. If this is the case, the auto-detected value is the CCSID that should be used when initializing the parser.

- Flags indicating which keyword values in the XML declaration were actually present.
- A flag to indicate how the auto-detected encoding value was determined. In certain cases, it's not possible to actually detect the encoding based on the bytes examined. In this case, the XML spec requires a parser to treat the document as if it were UTF-8 encoded, and this is what the QXD service will provide in the auto-detect value. A flag will be set in the flags field to indicate that the encoding was actually undetected, and that the encoding returned is the default UTF-8 value.
- The overall length of the XML declaration.

See “gxlpQuery — query an XML document” on page 80 or “GXL1QXD (GXL4QXD) — query an XML document” on page 138 for more details about how to acquire and use this data area.

gxlhxd.h (GXLYXD) - mapping of extended diagnostic area

This header (macro) contains the structure describing the extended diagnostic area that is returned when there is a failure in the z/OS XML parser. It is returned whenever the caller requests a control operation through the gxlpControl (GXL1CTL/GXL4CTL) service. The particular area that it is used to map depends on the control operation performed:

- *XEC_CTL_FIN (finish, and reset the parser) – this header (macro) maps the area pointed to directly by the ctl_data_p parameter of the gxlpControl (GXL1CTL/GXL4CTL) service.
- *XEC_CTL_FEAT (reset the parser with different features) – this header (macro) maps the area pointed to by the XFT_XD_PTR field of the GXLHXFT (GXLYXFT) structure.
- *XEC_CTL_LOAD_OSR (reset the parser and load an OSR for validation) – this header (macro) maps the area pointed to by the XOSR_XD_PTR field of the GXLHXOSR (GXLYXOSR) structure.

This mapping contains several types of key information that are of use for problem determination. Some of the more useful fields include the following:

- The address of the main parser anchor block. This is not generally useful for a caller, but is important for IBM service purposes.
- The input and output buffer addresses, and the current offsets into each. This shows which data the z/OS XML parser was processing at the time of the error.
- The size of the last memory allocation request made by the z/OS XML parser.
- Return and reason codes from the last memory allocation request made by the z/OS XML parser.
- Return and reason codes from system service exits (if exits are provided by the caller).
- Return code from the last request to switch to a specialty engine.
- A pointer to an area in the PIMA that is in the format of an output buffer, that contains enhanced error information for a validating parse when this information is requested.

gxlhxr.h (GXLYXR) - defines the return codes and reason codes

This contains all of the return and reason codes returned by z/OS XML. Each return and reason code has a descriptive comment. Also included is a reason code mask - *XRSN_REASON_MASK that is used to facilitate access to the low order 2 bytes of the reason code full word.

gxlhxsv.h (GXLYXSV) - mapping of the system service vector

Maps the area used to make assorted exit routines available to the z/OS XML parser. A complete description of the exits that can be specified and how to provide them can be found in Chapter 8, “z/OS XML System Services exit interface,” on page 145.

gxlhctl.h (GXLYCTL) - mapping of the control input parameters area

This header file and macro contain the various structures that are used in the gxlpControl (GXL1CTL/GXL4CTL) service. Each structure is used for a specific control call and passed to the control service on the ctl_data_p parameter. See the description of the ctl_data_p parameter in “gxlpControl — perform a parser control function” on page 54, or the ctl_data parameter in “GXL1CTL (GXL4CTL) — perform a parser control function” on page 114 for more details about the use of this structure.

gxlhxft.h (GXLYXFT) - mapping of the control feature input output area

This structure describes the area that is passed in to and back from the gxlpControl (GXL1CTL/GXL4CTL) service through the ctl_data_p (ctl_data) parameter. It is used to map this area when the caller is changing the parser feature settings by specifying the *XEC_CTL_FEAT value for the ctl_operation (ctl_option) parameter.

This structure includes an integer (fullword) value that contains the required features to reset. There are some features that cannot be reset, and which require that the parse instance to be terminated and re-initialized. This structure also contains the address of a fullword area in which the z/OS XML parser will place a pointer to the extended diagnostic area. This is the area that is mapped by gxlhxd.h (GXLYXD).

See the description of the ctl_data_p parameter in “gxlpControl — perform a parser control function” on page 54, or the ctl_data parameter in “GXL1CTL (GXL4CTL) — perform a parser control function” on page 114 for more details about the use of this structure.

gxlhxosr.h (GXLYXOSR) - mapping of the OSR control area

This structure describes the area that is passed in to and back from the gxlpControl (GXL1CTL/GXL4CTL) service through the ctl_data_p parameter. It should be used to map this area when the caller is loading an OSR for a validating parse by specifying the *XEC_CTL_LOAD_OSR value for the ctl_operation (ctl_option) parameter.

This structure holds the address of a buffer that contains the OSR, plus an optional name string that will be associated with the OSR. This name is currently optional, but it is recommended that every different OSR loaded be given a unique name. This can be useful for problem determination purposes in the event of an error.

This structure also contains the address of a fullword area in which the parser will place a pointer to the extended diagnostic area. This is the area that is mapped by gxlhxd.h (GXLYXD).

See the description of the ctl_data_p parameter in “gxlpControl — perform a parser control function” on page 54, or the ctl_data parameter in “GXL1CTL (GXL4CTL) — perform a parser control function” on page 114 for more details about the use of this structure.

gxlhosrg.h - OSR generator prototypes

This header contains includes for all of the OSR generator utility services, as well as the prototypes for those services. There are no Metal C or assembler macro versions of this header file.

gxlhosrd.h - mapping of the OSR generator diagnostic area

This header contains the structure that maps the extended diagnostic area returned from the OSR generator utility – similar to the way that gxlhxd.h (GXLYXD) describes the extended diagnostic area returned by the z/OS XML parser. Some of the more useful fields include the following:

- The address of the OSR generator Instance Memory Area (OIMA).
- The last return and reason code issued by the OSR generator.
- The last return and reason code issued by the StringID exit.
- An area containing a Java exception that may have been the cause of the failure. Some of the OSR generator is implemented in Java, so this area will contain the exception information when an error occurs in the Java code.

There are no Metal C or assembler macro versions of this header file.

gxlhxstr.h - StringID table

StringIDs are numeric values that are substituted for certain character strings that are encountered during the parse process. They can save space in the parsed data stream, and possibly improve performance if there are large numbers of repeated strings in the XML document being parsed. This can be the case with documents that make heavy use of namespaces with long URIs.

A caller may specify a StringID exit for the OSR generator to use, such that when a string is encountered, it will call the exit to either generate a new ID, if the string hasn't been seen before, or return an existing ID for strings which have been previously encountered. As the generator acquires these StringIDs, it saves them away in a table, and substitutes them for the strings that they represent within the OSR. The z/OS XML parser implements a similar behavior when it parses an XML document using StringIDs.

It will often be the case that the caller needs to use the same set of StringIDs at OSR generation time, and when a validating parse is performed with that OSR. The OSR generator API contains the gxluGenStrIDTable service that allows the caller to extract the StringID table from the OSR so that the table can be imported by the StringID exit used during the parse process. See “gxluGenStrIDTable — generate StringID table from an OSR” on page 100 for more details about how this service works.

This header file contains the structure definitions that describe the format of the StringID table that is exported from the OSR generator. The table is broken down into a fixed portion that contains information about the table, and a variable length portion containing the individual entries of the table. These are the structures that the StringID exit service can use to import the StringID table in preparation for a validating parse.

There are no Metal C or assembler macro versions of this header file.

Appendix E. Callable services examples - AMODE 31

GXL1CTL example

The following code calls the GXL1CTL service to change the feature bits for the z/OS XML parser. For the callable service, see “GXL1CTL (GXL4CTL) — perform a parser control function” on page 114. AMODE 64 callers use “GXL4CTL example” on page 225.

```
**********************************************************
* Setup parameter list to call GXL1CTL.                  *
*   Then call GXL1CTL.                                   *
**********************************************************
* Call GXL1CTL(PIMA,        (00)
*              CTL_Option,  (04)
*              CTL_Data,    (08)
*              Return_Code, (12)
*              Reason_Code) (16)
*
       LA    R9,SAMPLE_PIMA_PTR
       L     R9,0(R9)
       ST    R9,Parser_Parm
       SLR   R4,R4
       LA    R10,SAMPLE_CTL_OPTION
       ST    R10,Parser_Parm+4
       LA    R10,SAMPLE_CTL_DATA
       ST    R10,Parser_Parm+8
       LA    R10,SAMPLE_CTL_RC
       ST    R10,Parser_Parm+12
       LA    R10,SAMPLE_CTL_RSN
       ST    R10,Parser_Parm+16
*********************************************************
       LLGT  R15,CVTPTR
       L     R15,CVTCSRT-CVT(R15)
       L     R15,72(R15)
       L     R15,28(R15)
       LA    R1,Parser_Parm
       BALR  R14,R15
                    :
*********************************************************************
* Description of the SAMPLE Structure:
* *********************************************************************
SAMPLE                DSECT          Memory storage area
SAMPLE_HEADER         DS   0D
SAMPLE_EYE_CATCHER    DS   CL8       eye-catcher string
SAMPLE_RETCODE        DS   1F
SAMPLE_RSNCODE        DS   1F
SAMPLE_PIMA_PTR       DS   1F
SAMPLE_PIMA_LEN       DS   1F
SAMPLE_INIT_FEAT      DS   1F
SAMPLE_INIT_RC        DS   1F
SAMPLE_INIT_RSN       DS   1F
SAMPLE_CTL_OPTION     DS   1F
SAMPLE_CTL_DATA       DS   1F
SAMPLE_CTL_RC         DS   1F
SAMPLE_CTL_RSN        DS   1F
SAMPLE_TERM_RC        DS   1F
SAMPLE_TERM_RSN       DS   1F
SAMPLE_FLAGS1         DS   1F
SAMPLE_FLAGS2         DS   1F
SAMPLE_END            DS   0X
```

```
********************************************************************
NULL_Value            DC   1D'0'
CCSID                 DS   1F
PARSER_PARM           DS   8A
```

GXL1INI example

The following code initializes the PIMA and records the addresses of the caller’s system service routines (if any). For the callable service, see “GXL1INI (GXL4INI) — initialize a parse instance” on page 131. AMODE 64 callers use “GXL4INI example” on page 226.

```
**********************************************************
* Setup parameter list to call GXL1INI.                  *
*   Then call GXL1INI.                                   *
**********************************************************
* Call GXL1INI(PIMA,           (00)
*              PIMA_LEN,       (04)
*              CCSID,          (08)
*              Feature_Flags,  (12)
*              Sys_SVC_Vector, (16) Will be set to NULL
*              Sys_SVC_parm,   (20) Will be set to NULL
*              Return_Code,    (24)
*              Reason_Code)    (28)
*
       LA    R9,SAMPLE_PIMA_PTR
       L     R9,0(R9)
       ST    R9,Parser_Parm
       LA    R10,SAMPLE_PIMA_LEN
       ST    R10,Parser_Parm+4
       SLR   R4,R4
       LA    R10,XEC_ENC_IBM_037(R4)
       ST    R10,CCSID
       LA    R10,CCSID
       ST    R10,Parser_Parm+8
       LA    R10,SAMPLE_INIT_FEAT
       ST    R10,Parser_Parm+12
       LA    R10,NULL_Value
       ST    R10,Parser_Parm+16
       ST    R10,Parser_Parm+20
       LA    R10,SAMPLE_INIT_RC
       ST    R10,Parser_Parm+24
       LA    R10,SAMPLE_INIT_RSN
       ST    R10,Parser_Parm+28
*********************************************************
       LLGT  R15,CVTPTR
       L     R15,CVTCSRT-CVT(R15)
       L     R15,72(R15)
       L     R15,16(R15)
       LA    R1,Parser_Parm
       BALR  R14,R15
                    :
********************************************************************
* Description of the SAMPLE Structure:
* ********************************************************************
SAMPLE                DSECT           Memory storage area
SAMPLE_HEADER         DS   0D
SAMPLE_EYE_CATCHER    DS   CL8        eye-catcher string
SAMPLE_RETCODE        DS   1F
SAMPLE_RSNCODE        DS   1F
SAMPLE_PIMA_PTR       DS   1F
SAMPLE_PIMA_LEN       DS   1F
SAMPLE_INIT_FEAT      DS   1F
SAMPLE_INIT_RC        DS   1F
SAMPLE_INIT_RSN       DS   1F
SAMPLE_CTL_OPTION     DS   1F
```

```
SAMPLE_CTL_DATA       DS   1F
SAMPLE_CTL_RC         DS   1F
SAMPLE_CTL_RSN        DS   1F
SAMPLE_TERM_RC        DS   1F
SAMPLE_TERM_RSN       DS   1F
SAMPLE_FLAGS1         DS   1F
SAMPLE_FLAGS2         DS   1F
SAMPLE_END            DS   0X
********************************************************************
NULL_Value            DC   1D'0'
CCSID                 DS   1F
PARSER_PARM           DS   8A
```

GXL1PRS example

The following code parses a buffer of XML text and places the result in an output buffer. For the callable service, see “GXL1PRS (GXL4PRS) — parse a buffer of XML text” on page 135. AMODE 64 callers use “GXL4PRS example” on page 227.

```
*/**********************************************************************
*/   PARSE
*/**********************************************************************
*   CALL GXL1PRS(PIMA,OPTION_FLAGS,INBUF_PTR,INBUF_LEN,OUTBUF_PTR,
*       OUTBUF_LEN,RC,RSN);
         L     @02,PARM_PTR(,@03_PARM_PTR_PTR)
         L     @10,PIMA_PTR(,@02)
         ST    @10,@AL00001
         LA    @10,OPTION_FLAGS(,@02)
         ST    @10,@AL00001+4
         LA    @10,INBUF_PTR(,@02)
         ST    @10,@AL00001+8
         LA    @10,INBUF_LEN(,@02)
         ST    @10,@AL00001+12
         LA    @10,OUTBUF_PTR(,@02)
         ST    @10,@AL00001+16
         LA    @02,OUTBUF_LEN(,@02)
         ST    @02,@AL00001+20
         LA    @10,RC
         ST    @10,@AL00001+24
         LA    @02,RSN
         ST    @02,@AL00001+28
         OI    @AL00001+28,X'80'
         L     @10,CS$CVT
         L     @02,CS$CSRT+544(,@10)
         L     @10,CS$CSRFT+72(,@02)
         L     @15,GXLST31+20(,@10)
         LA    @01,@AL00001
         BALR  @14,@15
*   PARSE_RC = RC;
         L     @02,PARM_PTR(,@03_PARM_PTR_PTR)
         L     @10,RC
         ST    @10,PARSE_RC(,@02)
*   PARSE_RSN = RSN;
         L     @10,RSN
         ST    @10,PARSE_RSN(,@02)
*   END DO_PARSE;
*
```

GXL1TRM example

The following code releases all resources obtained (including storage) by the z/OS XML parser and resets the PIMA so that it can be re-initialized. For the callable service, see “GXL1TRM (GXL4TRM) — terminate a parse instance” on page 141. AMODE 64 callers use “GXL4TRM example” on page 228.

```
**********************************************************
* Setup parameter list to call GXL1TRM.                  *
*    Then call GXL1TRM.                                  *
**********************************************************
* Call GXL1TRM(PIMA,        (00)
*             Return_Code, (04)
*             Reason_Code) (08)
*
       LA    R10,SAMPLE_PIMA_PTR
       L     R10,0(R10)
       ST    R10,Parser_Parm
       LA    R10,SAMPLE_TERM_RC
       ST    R10,Parser_Parm+4
       LA    R10,SAMPLE_TERM_RSN
       ST    R10,Parser_Parm+8
*********************************************************
       LLGT  R15,CVTPTR
       L     R15,CVTCSRT-CVT(R15)
       L     R15,72(R15)
       L     R15,24(R15)
       LA    R1,Parser_Parm
       BALR  R14,R15
                   :
********************************************************************
* Description of the SAMPLE Structure:
* ********************************************************************
SAMPLE                DSECT           Memory storage area
SAMPLE_HEADER         DS   0D
SAMPLE_EYE_CATCHER    DS   CL8        eye-catcher string
SAMPLE_RETCODE        DS   1F
SAMPLE_RSNCODE        DS   1F
SAMPLE_PIMA_PTR       DS   1F
SAMPLE_PIMA_LEN       DS   1F
SAMPLE_INIT_FEAT      DS   1F
SAMPLE_INIT_RC        DS   1F
SAMPLE_INIT_RSN       DS   1F
SAMPLE_CTL_OPTION     DS   1F
SAMPLE_CTL_DATA       DS   1F
SAMPLE_CTL_RC         DS   1F
SAMPLE_CTL_RSN        DS   1F
SAMPLE_TERM_RC        DS   1F
SAMPLE_TERM_RSN       DS   1F
SAMPLE_FLAGS1         DS   1F
SAMPLE_FLAGS2         DS   1F
SAMPLE_END            DS   0X
********************************************************************
NULL_Value            DC   1D'0'
CCSID                 DS   1F
PARSER_PARM           DS   8A
```

Appendix F. Callable services examples - AMODE 64

GXL4CTL example

The following code calls the GXL4CTL service to change the feature bits for the z/OS XML parser. For the callable service, see “GXL1CTL (GXL4CTL) — perform a parser control function” on page 114. AMODE 31 callers use “GXL1CTL example” on page 221.

```
**********************************************************
* Setup parameter list to call GXL4CTL.                  *
*    Then call GXL4CTL.                                  *
**********************************************************
* Call GXL4CTL(PIMA,           (00)
*              CTL_Option,     (08)
*              CTL_Data,       (16)
*              Return_Code,    (24)
*              Reason_Code)    (32)
*
       LA    R9,SAMPLE_PIMA_PTR
       LG    R9,0(R9)
       STG   R9,Parser_Parm
       SLGR  R4,R4
       LA    R10,SAMPLE_CTL_OPTION
       STG   R10,Parser_Parm+8
       LA    R10,SAMPLE_CTL_DATA
       STG   R10,Parser_Parm+16
       LA    R10,SAMPLE_CTL_RC
       STG   R10,Parser_Parm+24
       LA    R10,SAMPLE_CTL_RSN
       STG   R10,Parser_Parm+32
*********************************************************
       LLGT  R15,CVTPTR
       L     R15,CVTCSRT-CVT(R15)
       L     R15,72(R15)
       LG    R15,64(R15)
       LA    R1,Parser_Parm
       BALR  R14,R15
                    :
********************************************************************
* Description of the SAMPLE Structure:
* ********************************************************************
SAMPLE                DSECT           Memory storage area
SAMPLE_HEADER         DS   0D
SAMPLE_EYE_CATCHER    DS   CL8        eye-catcher string
SAMPLE_RETCODE        DS   1F
SAMPLE_RSNCODE        DS   1F
SAMPLE_PIMA_PTR       DS   1D
SAMPLE_PIMA_LEN       DS   1F
SAMPLE_INIT_FEAT      DS   1F
SAMPLE_INIT_RC        DS   1F
SAMPLE_INIT_RSN       DS   1F
SAMPLE_CTL_OPTION     DS   1F
SAMPLE_CTL_DATA       DS   1F
SAMPLE_CTL_RC         DS   1F
SAMPLE_CTL_RSN        DS   1F
SAMPLE_TERM_RC        DS   1F
SAMPLE_TERM_RSN       DS   1F
SAMPLE_FLAGS1         DS   1F
SAMPLE_FLAGS2         DS   1F
SAMPLE_END            DS   0X
```

```
********************************************************************
NULL_Value            DC   1D'0'
CCSID                 DS   1F
PARSER_PARM           DS   16A
```

GXL4INI example

The following code initializes the PIMA and records the addresses of the caller's system service routines (if any). For the callable service, see "GXL1INI (GXL4INI) — initialize a parse instance" on page 131. AMODE 31 callers use "GXL1INI example" on page 222.

```
**********************************************************
* Setup parameter list to call GXL4INI.                  *
*    Then call GXL4INI.                                  *
**********************************************************
* Call GXL4INI(PIMA,           (00)
*              PIMA_LEN,       (08)
*              CCSID,          (16)
*              Feature_Flags,  (24)
*              Sys_SVC_Vector, (32) Will be set to NULL
*              Sys_SVC_parm,   (40) Will be set to NULL
*              Return_Code,    (48)
*              Reason_Code)    (56)
*
       LA    R9,SAMPLE_PIMA_PTR
       LG    R9,0(R9)
       STG   R9,Parser_Parm
       LA    R10,SAMPLE_PIMA_LEN
       STG   R10,Parser_Parm+8
       SLGR  R4,R4
       LA    R10,XEC_ENC_IBM_037(R4)
       ST    R10,CCSID
       LA    R10,CCSID
       STG   R10,Parser_Parm+16
       LA    R10,SAMPLE_INIT_FEAT
       STG   R10,Parser_Parm+24
       LA    R10,NULL_Value
       STG   R10,Parser_Parm+32
       STG   R10,Parser_Parm+40
       LA    R10,SAMPLE_INIT_RC
       STG   R10,Parser_Parm+48
       LA    R10,SAMPLE_INIT_RSN
       STG   R10,Parser_Parm+56
*********************************************************
       LLGT  R15,CVTPTR
       L     R15,CVTCSRT-CVT(R15)
       L     R15,72(R15)
       LG    R15,40(R15)
       LA    R1,Parser_Parm
       BALR  R14,R15
                    :
********************************************************************
* Description of the SAMPLE Structure:
* ********************************************************************
SAMPLE                DSECT           Memory storage area
SAMPLE_HEADER         DS   0D
SAMPLE_EYE_CATCHER    DS   CL8        eye-catcher string
SAMPLE_RETCODE        DS   1F
SAMPLE_RSNCODE        DS   1F
SAMPLE_PIMA_PTR       DS   1D
SAMPLE_PIMA_LEN       DS   1F
SAMPLE_INIT_FEAT      DS   1F
SAMPLE_INIT_RC        DS   1F
SAMPLE_INIT_RSN       DS   1F
SAMPLE_CTL_OPTION     DS   1F
```

```
SAMPLE_CTL_DATA      DS   1F
SAMPLE_CTL_RC        DS   1F
SAMPLE_CTL_RSN       DS   1F
SAMPLE_TERM_RC       DS   1F
SAMPLE_TERM_RSN      DS   1F
SAMPLE_FLAGS1        DS   1F
SAMPLE_FLAGS2        DS   1F
SAMPLE_END           DS   0X
********************************************************************
NULL_Value           DC   1D'0'
CCSID                DS   1F
PARSER_PARM          DS   16A
```

GXL4PRS example

The following code parses a buffer of XML text and places the result in an output buffer. For the callable service, see “GXL1PRS (GXL4PRS) — parse a buffer of XML text” on page 135. AMODE 31 callers use “GXL1PRS example” on page 223.

```
*/*******************************************************            */
*/*  DO_PARSE                                                           */
*/*******************************************************            */
*
*DO_PARSE:
*   PROCEDURE;
DO_PARSE STM   @14,@12,@SA00004
         STMH  @14,@12,@SH00004
*   CALL GXL4PRS(PIMA,OPTION_FLAGS,INBUF_PTR,INBUF_LEN,OUTBUF_PTR,
*       OUTBUF_LEN,RC,RSN);
         LG    @02,PARM_PTR(,@03_PARM_PTR_PTR)
         LG    @10,PIMA_PTR(,@02)
         STG   @10,@AX00001
         LA    @10,OPTION_FLAGS(,@02)
         STG   @10,@AX00001+8
         LA    @10,INBUF_PTR(,@02)
         STG   @10,@AX00001+16
         LA    @10,INBUF_LEN(,@02)
         STG   @10,@AX00001+24
         LA    @10,OUTBUF_PTR(,@02)
         STG   @10,@AX00001+32
         LA    @02,OUTBUF_LEN(,@02)
         STG   @02,@AX00001+40
         LA    @10,RC
         STG   @10,@AX00001+48
         LA    @02,RSN
         STG   @02,@AX00001+56
         L     @10,CS$CVT
         LLGTR @10,@10
         L     @02,CS$CSRT+544(,@10)
         LLGTR @02,@02
         L     @10,CS$CSRFT+72(,@02)
         LLGTR @10,@10
         LG    @15,GXLST64+48(,@10)
         LA    @01,@AX00001
         BASR  @14,@15
*   PARSE_RC = RC;
         LG    @02,PARM_PTR(,@03_PARM_PTR_PTR)
         L     @10,RC
         ST    @10,PARSE_RC(,@02)
*   PARSE_RSN = RSN;
         L     @10,RSN
         ST    @10,PARSE_RSN(,@02)
*   END DO_PARSE;
*
@EL00004 DS    0H
```

```
@EF00004 DS    0H
@ER00004 LMH   @14,@12,@SH00004
         LM    @14,@12,@SA00004
         BR    @14
```

GXL4TRM example

The following code releases all resources obtained (including storage) by the z/OS XML parser and resets the PIMA so that it can be re-initialized. For the callable service, see “GXL1TRM (GXL4TRM) — terminate a parse instance” on page 141. AMODE 31 callers use “GXL1TRM example” on page 223.

```
**********************************************************
* Setup paramter list to call GXL4TRM.                   *
*    Then call GXL4TRM.                                  *
**********************************************************
* Call GXL4TRM(PIMA,        (00)
*             Return_Code, (08)
*             Reason_Code) (16)
*
       LA    R10,SAMPLE_PIMA_PTR
       LG    R10,0(R10)
       STG   R10,Parser_Parm
       LA    R10,SAMPLE_TERM_RC
       STG   R10,Parser_Parm+8
       LA    R10,SAMPLE_TERM_RSN
       STG   R10,Parser_Parm+16
*********************************************************
       LLGT  R15,CVTPTR
       L     R15,CVTCSRT-CVT(R15)
       L     R15,72(R15)
       LG    R15,56(R15)
       LA    R1,Parser_Parm
       BALR  R14,R15
*********************************************************
                   :
********************************************************************
* Description of the SAMPLE Structure:
* ********************************************************************
SAMPLE                DSECT           Memory storage area
SAMPLE_HEADER         DS   0D
SAMPLE_EYE_CATCHER    DS   CL8        eye-catcher string
SAMPLE_RETCODE        DS   1F
SAMPLE_RSNCODE        DS   1F
SAMPLE_PIMA_PTR       DS   1D
SAMPLE_PIMA_LEN       DS   1F
SAMPLE_INIT_FEAT      DS   1F
SAMPLE_INIT_RC        DS   1F
SAMPLE_INIT_RSN       DS   1F
SAMPLE_CTL_OPTION     DS   1F
SAMPLE_CTL_DATA       DS   1F
SAMPLE_CTL_RC         DS   1F
SAMPLE_CTL_RSN        DS   1F
SAMPLE_TERM_RC        DS   1F
SAMPLE_TERM_RSN       DS   1F
SAMPLE_FLAGS1         DS   1F
SAMPLE_FLAGS2         DS   1F
SAMPLE_END            DS   0X
********************************************************************
NULL_Value            DC   1D'0'
CCSID                 DS   1F
PARSER_PARM           DS   16A
```

Appendix G. Exit examples

GXLEFRM (GXLFST example)

Restrictions: The following restrictions apply to this example:

- This sample was designed to be a basic example of a memory service exit, and was not designed with other system considerations in mind, such as the z/OS XML parser running in cross memory mode, SRB mode, or in a different key, for instance.
- This sample is not designed to work with any other service exits. The exit workarea is assumed to be used by this memory service exit only. (Note that both GXLGST and GXLFST services are considered as one service exit). As a result, this memory service exit can only work independently, with no other service exits running.

This sample frees an area of memory passed by the z/OS XML parser. For the exit service, see “GXLFST31 (GXLFST64) — free memory” on page 149. AMODE 31 callers use GXLE1FRM. AMODE 64 callers use GXLE4FRM.

Register 14 is used to store the return address, which must be kept intact in order to exit this subroutine correctly.

This example assumes register one, which is passed in from the caller, contains the address of the parameter list. The following input variables are used in the example:

SYS_SVC_PARM
Address of storage area that the caller of the z/OS XML parser wants to pass on to the exit.

MEMORY_LEN
Contain the length of the memory area requested to be free.

The following output variables are used in the example:

MEMORY_ADDR
The address of the memory to be freed.

EXIT_DIAG_CODE
Contains diagnostic information.

XSM_DC_INVALID_EYECATCHER_STR
Eye catcher is incorrect.

XSM_DC_FAIL_FREE_MEM31
Fail to release storage memory.

RETCODE

XSM_RC_FAILURE
Unable to free memory

XSM_RC_SUCCESS
The storage macro released the allocated memory successfully (greater than zero if deallocation failed).

EXIT_DIAG_CODE

XSM_DC_INVALID_EYECATCHER_STR
Eye catcher is incorrect.

XSM_DC_FAIL_FREE_MEM31
Fail to release storage memory.

GXLEGTM (GXLGST example)

Restrictions: The following restrictions apply to this example:

- This sample was designed to be a basic example of a memory service exit, and was not designed with other system considerations in mind, such as the z/OS XML parser running in cross memory mode, SRB mode, or in a different key, for instance.
- This sample is not designed to work with any other service exits. The exit workarea is assumed to be used by this memory service exit only. (Note that both GXLGST and GXLFST services are considered as one service exit). As a result, this memory service exit can only work independently, with no other service exits running.

This sample allocates an area of memory of the size requested by the z/OS XML parser. For the exit service, see "GXLGST31 (GXLGST64) — get memory" on page 147. AMODE 31 callers use GXLE1GTM. AMODE 64 callers use GXLE4GTM.

Register 14 is used to store the return address, which must be kept intact in order to exit this subroutine correctly.

This example assumes register one, which is passed in from the caller, contains the address of the parameter list. The following input variables are used in the example:

SYS_SVC_PARM
Address that was passed to the z/OS XML parser at initialization time.

MEMORY_LEN
Contains the length of the memory area requested by the z/OS XML parser.

The following output variables are used in the example:

MEMORY_ADDR
The address of the allocated memory.

EXIT_DIAG_CODE
Contains diagnostic information.

XSM_DC_INVALID_EYECATCHER_STR
Eye catcher is incorrect.

XSM_DC_INVALID_GET_MEM_LEN
Memory length is out of bound.

XSM_DC_FAIL_ALLOCATE_MEM31
Storage memory allocation failed.

RETCODE

XSM_RC_FAILURE
Unable to allocate memory.

XSM_RC_SUCCESS
The storage macro allocated the memory successfully (greater than zero if allocation failed).

GXLSYM example

Restrictions: The following restrictions apply to this example:

- This example was designed to be a basic example of a StringID service exit. It was not designed with other system considerations in mind, such as the z/OS XML parser running in cross memory mode, SRB mode, or in a different key, for instance.
- This example is not designed to work with any other service exits. The exit workarea is assumed to be used by this StringID service exit only. As a result, this StringID service exit can only work independently, with no other service exits running.

Note: This exit example is divided into the following 4 modules:

- "GXLEINI"
- "GXLEIDI (GXLSYM example module)" on page 232
- "GXLEIDR" on page 233

For the exit service, see "GXLSYM31 (GXLSYM64) — StringID service" on page 151. AMODE 31 callers use GXLSYM31. AMODE 64 callers use GXLSYM64.

GXLEINI

This example module does the following:

- Validates the caller specification and determines whether to use user defined or default values for storage size.
- Initializes all variables in XSI. (XSI is the data structure for the StringID sample exit).

Register 14 is used to store the return address, which must be kept intact in order to exit this subroutine correctly.

This example module assumes register one, which is passed in from the caller, contains the address of the parameter list. The following input variables are used in the example:

STRID_AREA_ADDR
Address of the XSI storage area.

STRID_AREA_LEN
Total length of the XSI Storage area.

STRID_MAX_NUM
The maximum number of StringIDs allowed.

SYM_MAX_SIZE
The maximum string length for each symbol.

The following output variables are used in this example module:

RETCODE

XSI_RC_FAILURE
If the storage area failed to initialize.

XSI_RC_SUCCESS
If the storage area successfully initialized.

DIAG_CODE
Contains diagnostic information.

XSI_DC_SYMBOL_STORAGE_TOO_SMALL
Storage size is too small.

GXLEIDI (GXLSYM example module)

This example module does the following:

- Search for an identical string in the tree.
- Inserts a string into a tree and returns a unique StringID. This is done as follows:
  1. Check first to make sure the length of the string is within the maximum symbol buffer size.
  2. Inserts the string into the root if the tree is empty or searches down the tree to find the appropriate empty leaf node.
  3. When the insert node location is found, its address will be passed to the INSERT_STRING subroutine. The subroutine will create a new leaf node and then insert the string.
  4. Return the StringID if the string inserted successfully.

Note: This is the actual exit pointed to in the SYS_SVC_VECTOR table.

Register 14 is used to store the return address, which must be kept intact in order to exit this subroutine correctly.

This example module assumes register one, which is passed in from the caller, contains the address of the parameter list. The following input variables are used in the example module:

SYS_SVC_PARM
Address of storage area that the caller of the z/OS XML parser wants to pass to the exit. It also contains the XSI structure information.

STR The string that will be inserted into the tree.

STRLEN
Length of the current string needed to be inserted. Length is derived from the number of bytes of the characters in the string.

CCSID Identifier for the string's character set.

The following output variables are used in the example module:

STRID The index of the inserted or found string.

EXIT_DIAG_CODE
Contains diagnostic information.

XSI_DC_INCORRECT_PARM_STRLEN
String length is out of bound.

XSI_DC_OUT_OF_STORAGE_SPACE
Allocated storage is full.

XSI_DC_INCORRECT_EYE_CATCHER
Eye catcher is incorrect.

XSI_DC_MAX_OUT_ID_LIST_ENTRIES
StringID list is full.

RETCODE

XRC_FAILURE
Failed to insert or search for *STR*.

XRC_SUCCESS
String was inserted or found.

GXLEIDR

This example module uses the input StringID to access a table and returns the address and length of the string associated with the StringID. The string is saved in the storage pointed to by SYS_SVC_PARM during the initialization of the parser (GXLINI) and in StringID processing (GXLEINI).

Register 14 is used to store the return address, which must be kept intact in order to exit this subroutine correctly.

This example module assumes register one, which is passed in from the caller, contains the address of the parameter list. The following input variables are used in the example module:

SYS_SVC_PARM
Address of storage area that the caller of the z/OS XML parser wants to pass to the exit. It also contains the XSI structure information.

STRID StringID used for indexing the list.

The following output variables are used in the example module:

STR_ADDR
Address of string from requested StringID.

STRLEN
The length of the string found by StringID.

DIAG_CODE
Contains diagnostic information.

XSI_DC_INCORRECT_StringID_OUTOFBOUND
STRID length is out of bound.

XSI_DC_INCORRECT_ID_LOCATION_ERROR
StringID does not match.

XSI_DC_INCORRECT_EYE_CATCHER
Eye catcher is incorrect.

RETCODE

XSI_RC_FAILURE
The string cannot be retrieved.

XSI_RC_SUCCESS
The string was retrieved successfully.

GXLESTRI

Restrictions: The following restrictions apply to this example:

- This sample was designed to be a basic example of a StringID service exit, and was not designed with other system considerations in mind, such as the z/OS XML parser running in cross memory mode, SRB mode, or in a different key, for instance.
- This sample is not designed to work with any other service exits. The exit workarea is assumed to be used by this service exit only. As a result, this memory service exit can only work independently, with no other service exits running.

This sample does the following:

- Initializes the structure (referred to herein as XSI).
- Searches for a string in the list and then returns its ID if the string is found.
- Inserts new strings.
- Validates memory requirements based on user input.
- Defines the default values.
- Initializes all variables in XSI.

The purpose of this StringID service exit routine is to demonstrate how a combination of Language Environment/Metal C StringID service exits could be written for the z/OS XML parserand the OSR generator.

Guidelines for using this exit with the z/OS XML parser: When this StringID service exit routine is used as an exit to the z/OS XML parser, the following guidelines apply:

- A prolog and epilog are required. This is used to set up DSA linkage. More details are below.
- The work area must be large enough to accommodate a DSA at the head of the work area, along with space for the stack frames. (This sample contains a main routine, a few local variables, and calls a subroutine on the first call for initialization.)
- The work area and immediately following the DSA and the stack space contains the storage that will be mapped to the XSI structure.

Guidelines for using this exit with the OSR generator: When this StringID service exit routine is used as an exit to the OSR generator, the following guidelines apply:

- A prolog and epilog are NOT required.
- The main routine name must be exported.
- The entire work area is mapped to the XSI structure defined here and must be large enough to accommodate this.

The user must pass in the value of available storage space for the XSI structure to this exit via the storage_size member in the XSI structure. Here is an example, when using the exit for the z/OS XML parser:

```
// Allocate storage.
stringIDArea = malloc(40960);
// Clear storage.
memset(stringIDArea,0,40960);
// Adjust pointer past DSA/frame(s) to beginning
// of storage available to the XSI structure.
ptr_val = (unsigned long)(stringIDArea);
ptr_val += XSI_DSA_SPACE;
XSI xsi_ptr = (XSI)(ptr_val) ;
```

```
// Set storage size and adjust for DSA/frame(s).
xsi_ptr->storage_space = 40960;
xsi_ptr->storage_space -= XSI_DSA_SPACE;
```

When using the exit for the OSR generator, use the following example:

```
// Allocate storage.
stringIDArea = malloc(40960);
// Clear storage.
memset(stringIDArea,0,40960);
XSI xsi_ptr = (XSI)(stringIDArea) ;
// Set storage size.
xsi_ptr->storage_space = 40960;
```

Register 14 is used to store the return address, which must be kept intact in order to exit this subroutine correctly.

This example assumes register one, which is passed in from the caller, contains the address of the parameter list. The following variables are used in the example:

SYS_SVC_PARM
A pointer to the address of the storage to be used for this exit.

STRING
The string passed in from the OSR generator.

STR_LEN
The value of the string length passed in from the OSR generator.

STRINGID
The value of the string ID set by this exit.

CCSID The Coded Character Set Identifier passed in from the OSR Generator.

DIAG_CODE
The diagnostic code set by this exit.

RETURN_CODE
The return code set by this exit.

A description of the XSI structure is provided below. The XSI structure includes the XSI header and StringID array list.

EYE_CATCHER
The eye catcher for this structure. Used to confirm initialization.

VERSION
The version number for this exit.

STORAGE_SPACE
The value of the size of storage allocated for this exit.

DIAG_CODE
The diagnostic code set by the exit.

NEXT_ID
The next available value for the StringID.

INDEX
The index of the next XSI_NODE to update.

STRINGLIST
An array of XSI_NODE that is populated with the StringID, strings and their lengths.

A description of the prolog and epilog is provided below:

```
Prolog for AMODE 31, Changes for AMODE 64 are in parenthesis
        ST(G)M   14,12,12(13)    Save entry regs in callers area
        L(G)     15,0(1)         Load address of Users storage
        L(G)     15,0(15)        Load the actual storage
        ST(G)    15,8(,13)       Save caller DSA in prev
        ST(G)    13,4(,15)       Save current DSA in callers
        LR(G)    13,15           Set start of storage to DSA
        MEND
Epilog for AMODE 31, Changes for AMODE 64 are in parenthesis
        L(G)     13,4(13)        Load the previous DSA
        LM(G)    14,12,12(13)    Restore the registers
        BR       14              Return
        MEND
```

Appendix H. CICS examples

The example below shows how to define GXLINPLT to the CICS CSD:

```
//GXLCSD   JOB <your jobcard>
//*****************************************************************/
//* Function:                                                     */
//*                                                               */
//* This is the sample of the DFHCSDUP job to create the resource */
//* definitions required for XML System Services.                 */
//* It defines one resource group:                                */
//* GXLXMLCG contains definitions needed for XML system services  */
//* The user must install group GXLXMLCG, it is recommended to    */
//* add group GXLXMLCG to the current grouplist for the CICS      */
//* region or add to grouplist GXLXMLCL as is shown below.        */
//* Before using this sample job replace the default parameter    */
//* values with the values of your CICS installation.             */
//*                                                               */
//*****************************************************************/
//*- - SET SYMBOLIC PARAMETERS                                   -*/
//*
//SETCID  SET CID='CICS410.CICS' ! Qualifier for CICS library
//SETCSD  SET CSD='TTCICS4.CICS' ! Qualifier for target CICS CSD
//DFHCSDUP EXEC PGM=DFHCSDUP,REGION=4M
//STEPLIB  DD DISP=SHR,DSN=&CID..SDFHLOAD
//DFHCSD   DD DISP=SHR,DSN=&CSD..DFHCSD
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
*
* delete the group GXLXMLCG
*
DELETE  GROUP(GXLXMLCG)
*
* Add the group to GRPLIST GXLXMLCL
*
*
ADD GROUP(GXLXMLCG) LIST(GXLXMLCL)
*
*
*
* Programs
*  Define GXLINPLT as a program in group GXLXMLCG
*
  DEFINE PROGRAM(GXLINPLT) GROUP(GXLXMLCG) LANGUAGE(ASSEMBLER)
  DESCRIPTION(XML PLT Program required by XML System Services)
  CONCURRENCY(QUASIRENT) DATALOCATION(ANY) EXECKEY(USER)
*
* Note that CONCURRENCY(QUASIRENT) ensures that the program
*  runs under the CICS QR TCB, which is what we want.
/*
//
```

Below is a sample job to update the PLT table:

```
//SUIMGLQ  JOB  <JOBCARD>
//*
//TABLEASM EXEC DFHAUPLE
//ASSEM.SYSUT1 DD *
        TITLE 'DFHPLTI1 - ADD GXLINPLT TO PLT TABLE'
***********************************************************************
*                                                                     *
* MODULE NAME = DFHPLTI1                                              *
*                                                                     *
* DESCRIPTIVE NAME = LIST OF PROGRAMS TO BE EXECUTED DURING CICS      *
*                    SYSTEM INITIALIZATION                             *
*                                                                      *
*                                                                      *
* FUNCTION =                                                           *
*                                                                      *
*   THIS LIST SPECIFIES THE GXLXMLCG GXLINPLT PROGRAM TO BE EXECUTED   *
*   DURING CICS TS SYSTEM INITIALIZATION SO GXLIMODV GETS LOADED       *
*   AT THE RIGHT LEVEL SO IT CAN BE USED BY TRANSACTION PROGRAMS.      *
*   THIS PROGRAM REQUIRED SYSTEM INITIALIZATION PARAMETER              *
*   PLTPI=I1.                                                          *
*                                                                      *
*
*----------------------------------------------------------------------*
************************************************************************
*
   DFHPLT TYPE=INITIAL,SUFFIX=I1
*
*
*  PROGRAMS SPECIFIED BEFORE THE DFHDELIM PROGRAM  ARE RUN
*  DURING SECOND INITIALIZATION STAGE.
*  PROGRAMS SHOULD ALSO BE DEFINED TO CICS BY DFHCSDUP OR RDO
*
   DFHPLT TYPE=ENTRY,PROGRAM=DFHDELIM
*
*  PROGRAMS THAT SHOULD BE RUN IN THE THIRD INITIALIZATION
*  PHASE (IF ANY) CAN BE SPECIFIED BELOW.
*  PROGRAMS SHOULD ALSO BE DEFINED TO CICS BY DFHCSDUP OR RDO
*
   DFHPLT TYPE=ENTRY,PROGRAM=GXLINPLT
*
   DFHPLT TYPE=FINAL
*
END
```

Appendix I. Supported encodings

The following table displays the encodings supported by z/OS XML System Services. Displayed in the table are the code page names, associated CCSID and equate names. Assembler callers use equate names without the "GXLH" prefix.

Rule: If you require a different encoding, you must first convert to one of the below before invoking the z/OS XML parser.

Table 38. Code page CCSID values

Code page	CCSID	Equate Names
UTF-8	1208	GXLHXEC_ENC_UTF_8
UTF-16 (big endian)	1200	GXLHXEC_ENC_UTF_16
EBCDIC/IBM-037	37	GXLHXEC_ENC_IBM_037
EBCDIC/IBM-273	273	GXLHXEC_ENC_IBM_273
EBCDIC/IBM-277	277	GXLHXEC_ENC_IBM_277
EBCDIC/IBM-278	278	GXLHXEC_ENC_IBM_278
EBCDIC/IBM-280	280	GXLHXEC_ENC_IBM_280
EBCDIC/IBM-284	284	GXLHXEC_ENC_IBM_284
EBCDIC/IBM-285	285	GXLHXEC_ENC_IBM_285
EBCDIC/IBM-297	297	GXLHXEC_ENC_IBM_297
EBCDIC/IBM-500	500	GXLHXEC_ENC_IBM_500
EBCDIC/IBM-871	871	GXLHXEC_ENC_IBM_871
EBCDIC/IBM-1047	1047	GXLHXEC_ENC_IBM_1047
EBCDIC/IBM-1140	1140	GXLHXEC_ENC_IBM_1140
EBCDIC/IBM-1141	1141	GXLHXEC_ENC_IBM_1141
EBCDIC/IBM-1142	1142	GXLHXEC_ENC_IBM_1142
EBCDIC/IBM-1143	1143	GXLHXEC_ENC_IBM_1143
EBCDIC/IBM-1144	1144	GXLHXEC_ENC_IBM_1144
EBCDIC/IBM-1145	1145	GXLHXEC_ENC_IBM_1145
EBCDIC/IBM-1146	1146	GXLHXEC_ENC_IBM_1146
EBCDIC/IBM-1147	1147	GXLHXEC_ENC_IBM_1147
EBCDIC/IBM-1148	1148	GXLHXEC_ENC_IBM_1148
EBCDIC/IBM-1149	1149	GXLHXEC_ENC_IBM_1149

Appendix J. Enabling z/OS V1R12 XML functionality in z/OS V1R10 and z/OS V1R11

Functionality was added to z/OS XML System Services in z/OS 1.12 that is available in z/OS 1.10 and z/OS 1.11 with APAR OA32251; PTFs UA59081 and UA59082. This APAR includes support for schema discovery, parsing of XML document fragments and restrict root support.

Schema discovery enhances the usability of the validating parser by allowing the caller to query the XML document schema locations detailed in the “schemaLocation” and “noNamespaceSchemaLocation” attributes, in addition to the root element namespace and local name. Following this, the caller will have the opportunity to load an OSR without having to reset the parse. See “Obtaining information on schema locations” on page 19for more information on schema discovery support.

Parsing of document fragments without obtaining and parsing an entire document is now supported when parsing in z/OS XML System Services with schema validation. . See “Parsing XML document fragments with validation” on page 17for more information on fragment parsing.

Restrict root support allows an z/OS XML System Services caller to restrict the root name against a given root element name or a list of root element names when performing a validating parse. See “Restricting the root element name” on page 16for more information on restrict root support.

To enable the support in z/OS 1.10 and z/OS 1.11 environments, the caller must complete the following steps:

1. Load GXLIMOD2, the alternate validating parser into memory for use by the application.
   - The alternate parser, GXLIMOD2, contains support for the z/OS 1.12 XML System Services functions. To load the z/OS 1.12 version of the validating parser using the GXL1LOD(GXL4LOD) API, specify XEC_LOD_VPARSE_ALT for the function_code. See “gxlpLoad — load a z/OS XML function” on page 76and “GXL1LOD (GXL4LOD) — load a z/OS XML function” on page 142for more information on loading a validating parser.
2. Parse with an OSR that supports the full z/OS 1.12 functionality.
   - An OSR generated on a z/OS 1.12 system can be used on a z/OS 1.10 or z/OS 1.11 system with APAR OA32251 installed and GXLIMOD2 loaded. This OSR will fully support all the functions listed above.
   - To generate an OSR on a z/OS 1.10 or z/OS 1.11 system that fully supports the z/OS 1.12 functionality listed above using the **xsdosrg** command, specify the **–a** option on the command. OSRs that were generated on z/OS 1.10 and z/OS 1.11 systems without the **–a** option can be used with the alternate parser, but will not fully support all the new functions listed above.
   - To generate an OSR on a z/OS 1.10 or a z/OS 1.11 system that fully supports the z/OS 1.12 functionality listed above using the C interface, specify GXLHXEC_OSR_ALT for the feature_flags field on the gxluInitOSRG interface.

- To generate an OSR that supports the full z/OS 1.12 functionality using the Java interface, specify type=OSRINI_ALT on the newOSRGenerator method in the gxlOSRGenerator class, when generating an OSR.

The following is a list of examples:

- C example of loading the alternate validating parser (GXLIMOD2)

```
int f_code = GXLHXEC_LOD_VPARSE_ALT;
int f_data = 0;
int lodRet = 0;
int lodRsn = 0;
/* load the alternalte parser */

gxlpLoad(f_code,
         f_data,
         &lodRet,
         &lodRsn);
```

- **xsdosrg** example of generating a z/OS 1.12 functional level OSR

```
xsdosrg -a -o test.osr test.xsd
```

- C example of generating a z/OS 1.12 functional level OSR

```
void * myOIMA = NULL;    /* OSR generator instance memory area */
long   myOIMALength = GXLHXEC_MIN_OIMA_SIZE; /* length of OIMA */
void * sysSvcWorkarea = NULL;
featureFlags = GXLHXEC_OSR_ALT; /* Alternate OSR requested     */
int localRC = 0;
int localRSN = 0;

myOIMA = malloc(GXLHXEC_MIN_OIMA_SIZE);
localRetVal = gxluInitOSRG(myOIMA,
                           myOIMALength,
                           featureFlags,
                           sysSvcWorkarea,
                           &localRC,
                           &localRSN);
```

- Java example of generating a z/OS 1.12 functional level OSR

```
/* When issuing newOSRGenerator, specify type =
gxlOSRGenerator(gxlOSRGenerator.OSRINI_ALT) */

/* This will tell subsequent calls to generate an alternate osr */


myOSRGen = gxlOSRGenerator.newOSRGenerator(gxlOSRGenerator.OSRINI_ALT);
```

Appendix K. Accessibility

Accessible publications for this product are offered through the z/OS Information Center, which is available at www.ibm.com/systems/z/os/zos/bkserv/.

If you experience difficulty with the accessibility of any z/OS information, please send a detailed message to mhvrcfs@us.ibm.com or to the following mailing address:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the z/OS Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1!

(KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Note:

  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.
- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (http://www.ibm.com/software/support/systemsz/lifecycle/)
- For information about currently-supported IBM hardware, contact your IBM representative.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

DB2® is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Index

Product Number: 5650-ZOS

Printed in USA

SA38-0681-00