OPEN UTM

Global Challenge. Universal Solution.

XML for openUTM

Version 3.0A50

1 Contents

2 Introduction

2.1 XML in openUTM

A common problem facing data transfer in distributed applications is that the information about the structure and meaning of the data must be available at multiple points and must also be identical: An application creates a data packet in a certain format and sends it to a remote application which must have precise knowledge of its structure and content in order to interpret the data correctly.
This is where the transfer of data in the form of XML documents can deliver great benefits: Information about a data element is supplied in the document itself; the elements are identified by their names and not by their precise placement in the document. Thus, changes to the data do not necessarily result in changes needing to be made in the processing program at the point where the data is received. Moreover, XML documents are platform-independent, as all the contents are available in printable form.

2.2 What is XML?

XML (eXtensible Markup Language) is a subset of SGML (Standard Generalized Markup Language) and provides EDP users with a meta language that offers a host of application possibilities. The [XML specification] published by the W3C defines the format, structure and possible content of an XML document.
Everyone now has the opportunity, e.g. using a DTD (Document Type Definition) or an XML Schema (see section 3.10 XML Schema Validation), to define their own language syntax, i.e. they can restrict the namespace of elements, attributes and element contents and describe relationships.
Special programs (called XML parsers) are available that can validate a document. In other words, a parser can check whether a document is a "well-formed " XML document as defined in the XML specification or, if a DTD or an XML schema is present, whether it is a "valid" document according to this DTD or schema. A parser can be used to convert and process an XML document of this type into an object tree conforming to DOM. DOM (Document Object Model) is an abstract API for creating, reading and processing object trees of this type (see [DOM specification]).

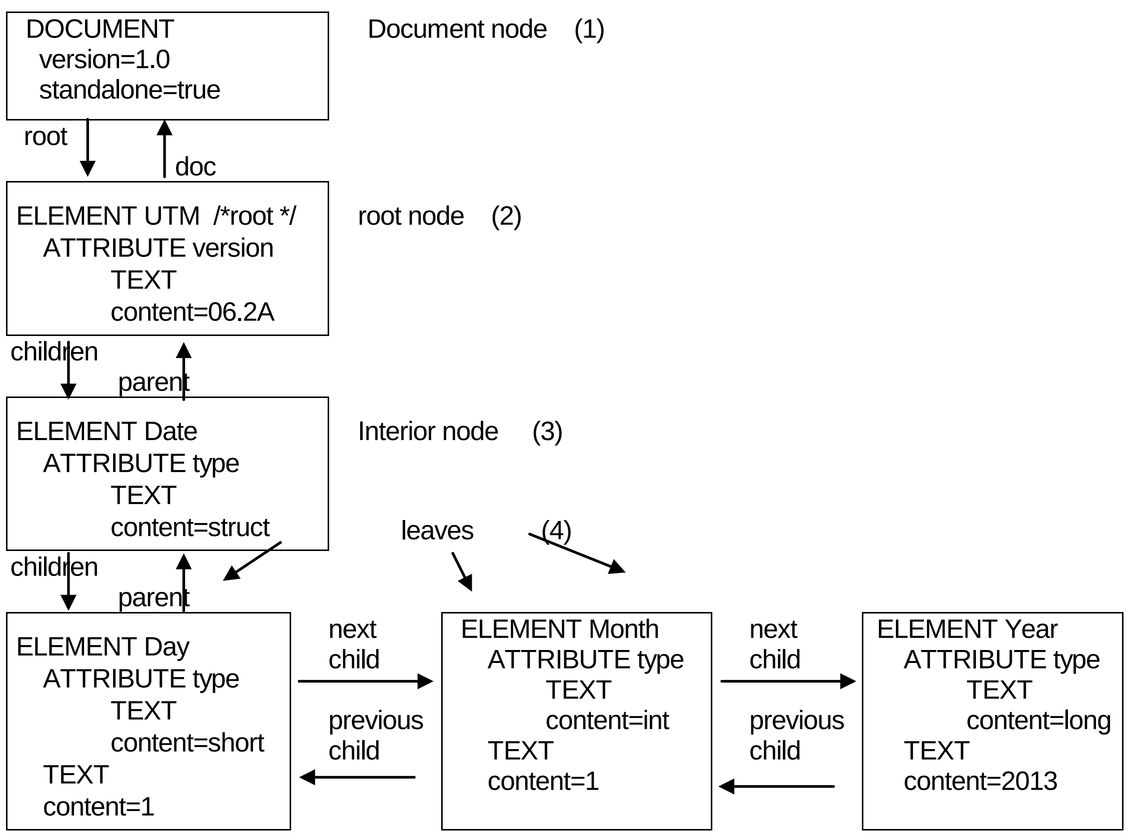
2.3 Definitions

Some key terms are explained or defined below.
An **XML** document is a document that is well-formed, as defined in the [XML specification]. For other concepts such as element, tag, attribute, see the [XML specification].
An (XML) object is an object tree which, according to DOM, corresponds to an XML document and on which it is possible to operate by means of an API (see [DOM specification]). An XML object is referred to below simply as an object, and an XML document simply as a document.
In an object, nested elements of a document are arranged hierarchically in a tree-like structure. Starting from a node, called the root, there exist child nodes, which can have children in their turn. Child nodes with the same higher-level node, called the parent, are linked with one another and are called siblings. Nodes with no children are called leaf nodes or leaves, while nodes with children are called interior nodes.
Internally, a node, also referred to below as an element, consists of multiple nodes of different types (element node, attribute node, text node).
Creating a document from an object and, vice versa, an object from a document is called **parsing** or **conversion**.
All nodes below an interior node are called, together with this node, a **subobject** or **subtree**.

Thus, for example, the document:

```
<?xml version="1.0"?>                    corresponds to node (1)
<UTM version="06.2A">                    corresponds to node (2)
  <Date type="struct">                   corresponds to node (3)
    <Day type="short">1</Day>            corresponds to node (4)
    <Month type ="int">1</Month>         corresponds to node (4)
    <Year type ="long">2013</Year>       corresponds to node (4)
  </Date>                                corresponds to node (3)
```

corresponds to the following object:



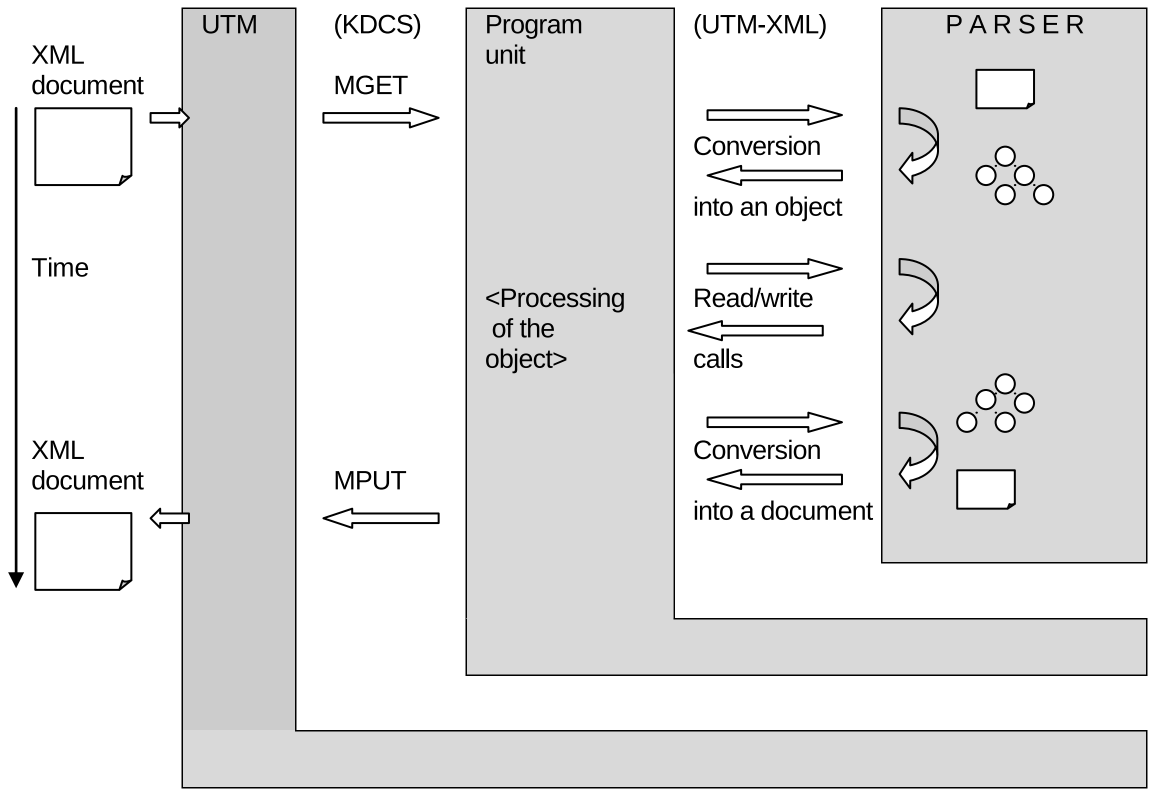
Nodes (2) and (3) without (4) do not form e.g. a complete subtree. Nodes (3) and (4) form a subtree of the object, node 'Day' is a sibling node of 'Month' and 'Year'.
In element <Month type="int">1</Month>,

-	<Month type="int">	is the start tag,
-	Month	is the tag or element name and
-	1	is the value of the element.
-	type="int" and	is the attribute with 'type' as attribute name and 'int' as attribute value
-	</Month>	is the end tag.

2.4 Potential applications in the openUTM environment

In the following sections, the server is referred to as the sender (as creator of the XML document) and the client as the recipient (as reader of the XML document).

In distributed applications, a server can provide e.g. an XML document conforming to a C or COBOL data structure. It creates the XML object, always first creating an empty XML object. The XML object can then be built up using write functions. At the end, a function is called that converts the XML object into an XML document. This can then be sent to the recipient by means of the sender-specific communication functions (KDCS, CPIC, Socket, etc.).
The recipient reads the entire document with the recipient-specific communication functions and converts the XML document into an XML object. He can then read or modify the elements relevant to him via the tag names (corresponding to the structure elements) or create new elements which (converted into an XML document, sent and reconverted into an XML object) can, in turn, be processed.
Extensions to the structures or to the document do not change access to already existing elements. Thus, for example, the server providing the extended structures is recompiled first. Then, the clients requiring the extensions can follow in succession. If an element is removed, the clients would be converted first, and finally the server.
Taking openUTM as client as an example, the processing sequence for XML documents therefore looks something like this:



With openUTM as server, the following actions, for example, are executed to create a document:

- Read relevant data (MGET),
- Create an empty (new) object (KXLCreateNewObj),
- Create the object with write calls (KXLWrite),
- Convert into a document (KXLConvObjToDoc) and
- Send the XML document (MPUT)

Integration of the interface in an openUTM client opens up further application options.

3 Program interface UTM-XML

3.1 New functions in UTM-XML V3.0

The following new functions are offered in UTM-XML V3:

XML schema support

UTM XML offers the following new functions to support XML schema functionality:

KXLConvDocToObjAndValid	to convert an XML document with schema validation
KXLParseSchema	to parse an XML schema in memory
KXLParseSchemaFile	to parse an XML schema file
KXLValidDoc	to validate an XML object against a schema
KXLValidDocBuf	to validate an XML document in memory against a schema
KXLFreeSchema	to free the schema memory areas
KXLSchemaGetRoot	to get the root node of the schema object (only in Cobol)

Interface initialization:

KXLInitEnv	to initialize the interface environment

The following new functions are offered in UTM-XML V3.0A40:

KXLFindNode	to read an XML element without regard to namespaces

The following new functionality is offered in UTM-XML V3.0A50:

Support for NetCOBOL compiler

On Unix and Windows systems, it is also possible to compile Cobol programs using the NetCOBOL compiler from Fujitsu.

Support for the Visual COBOL compiler

On Unix and Windows systems, it is also possible to compile Cobol programs using the Visual COBOL compiler from Micro Focus.

3.2 Coverage of the API

The present UTM-XML API is a programming interface which can be used to process an object tree conforming to DOM with access functions. Among other things it allows C and Cobol data structures to be stored and processed as XML objects. Subject to certain preconditions, user-defined element types can also be processed.
Access functions are available for C, C++ and Cobol. They are executable on BS2000, UNIX, Linux and Windows platforms.
The parser on which the API is based is the [GNOME parser] libxml2.
Use of the functions is illustrated in the Appendix with the aid of a comprehensive example.

The API is divided into several areas:

- Conversion of data types into the t_value structure and back (KXLFromXxx, KXLToXxx, where Xxx = Short, Int, Long, Float, Double, Char, String, Struct, and KXLFromArray)
- Creation of XML objects (KXLCreateNewObj, KXLWrite)
- Navigation of XML objects (KXLSetSubObject, KXLSetRootNode, KXLSetParentNode)
- Reading XML objects (KXLRead, KXLReadNode, KXLReadNextSib, KXLReadChild, KXLReadNextSingleNode, KXLReadAttr),
- Conversion of XML objects into XML documents and vice-versa, and releasing objects (KXLConvDocToObj, KXLConvObjToDoc, KXLFreeObj),
- Character set handling (KXLGetHomeEnc, KXLGet/SetDocEnc, KXLStringFromUTF8, KXLStringToUTF8),
- Namespace management (KXLWriteNS, KXLDelNS, KXLReadNSList, KXLSearchNS),
- Validation functions (KXLConvDocToObjAndValid, KXLParseSchema,
- KXLParseSchemaFile, KXLValidDoc, KXLValidDocBuf, KXLFreeSchema)
- Initialization and Diagnostic functions (KXLInitEnv, KXLTSENV, KXLGetLastParserError) and
- Other functions (KXLGetSizeofNodelist)

3.3 Parameter passing

All the contents of the elements stored in the XML object with the UTM-XML API are transferred as printable character strings; this means, for example, that int numeric values must first be converted into character strings before being written (KXLFromInt), and possibly reconverted (KXLToInt) for further processing after being read. In order to preserve the type information, each value is passed as a structure of two pointers which point to the type of the value and the value as a character string. Thus, the content of an element is always represented in the following format:

```
struct t_value
        {char*   pType;
         char*   pValue;
        };
```

where pType contains the address of a character string containing the type of the element, and pValue the address of the (printable) element value. Thus, for example, the `float` value `1.45` is passed as a structure of two pointers which point to the strings "float" and "1.45".

As attributes are accessed using the same functions as elements, the contents of attributes are expected and delivered by the t_value structure in the ‘p_value’ field. The ‘pType’ field points to an empty character string in each case.

3.4 Mapping the (C) data structures onto XML documents

When a (C) data structure is mapped onto an XML document, the hierarchy of the structure is preserved. Thus, higher-level structures are represented as enclosing elements of the XML document (or as parent in the object tree), and subordinate structures or elementary data as enclosed elements (or as child nodes in the object tree).

Example: The C data structure

```
typedef struct Sdate
{ short    Day;
  int      Month;
  long     Year; };
```

```
Sdate Date={1,1,2013}
```
is represented as follows in an XML document under the UTM-XML API:
```
<Date type="struct">
        <Day type="short">1</day>
        <Month type ="int">1</month>
        <Year type ="long">2013</year>
</Date>
```
The object tree contains the node named 'Date' with the child nodes 'Day', 'Month' and 'Year'.
The type attributes can be omitted, in which case 'type="struct"' is implicitly assumed for interior nodes and 'type="string"' for leaf nodes.

3.5 Creating an XML document/XML object

An object tree is created either by parsing an XML document (KXLConvDocToObj) or is built up so that an XML document can be generated from it. For this purpose, a new object must be created first (KXLCreateObj) and the individual nodes written with KXLWrite. Read, write and delete actions can be executed in the object thus created. Access to subtrees can be optimized by positioning the pointers in the object tree (KXLSet...). Finally, the object is converted into an XML document with KXLConvObjToDoc and can be stored in the data storage system (e.g. database) or sent to other communication partners.

All the data that the XML object is to contain must be written sequentially by means of a KXLWrite for each individual value. From the initially empty XML object, this results in a tree-like structure consisting of element nodes, called the **object tree**, with each structure and each array represented by an "interior" node. The associated subobject, or subtree, contains the components of the structure or array. Each single data item (of type short, int, long, float, double, char) and each character string (string) is represented by a single node, or leaf. The full name of each element is composed of the names of all nodes that lie between the root node and the particular node in the object tree.

When a node of the XML object is accessed, a pointer must be specified to a node which does not necessarily have to be the root node of the XML object. By positioning with KXLSetSubObject or when writing a node of type `struct` or `array`, the user receives the address of a node within the XML object, from which a subtree can be created or read without needing to specify the full name of an element in each call and running through the full name path.

3.6 Character set

The character set of the element and attribute names and their content are described in the [XML specification]. A greatly simplified description which lays no claim to completeness and correctness is given below. In case of doubt, the [XML specification] is always authoritative.

1. Attribute and tag names may contain letters, digits, ".", "-", "_" and ":", but must not begin with a digit, "." or "-".

2. The value of an attribute may, if enclosed in double quotes ("), contain all characters except "%", "&", """ (double quote) and no references of type "reference" (see point 3.); if enclosed in single quotes ('), it may contain all characters except "%", "&", "'" (single quote) and no references of type "reference" (see point 3.).

3. The contents of an element consist of an arbitrary mix of

- elements	
- references	such as &name; or &#nn; or &#xnn;, where 'name' is structured as described in 1. and 'nn' is a numeric character string.
- CDSects	such as <![CDATA[charData]]>
- PIs	such as <? PI-Target charData ?>
- comments	such as <!- - charData - - >

- charData, which may contain all characters except "<", "&" and the particular terminating character string of the entity, such as]]>, ?> or -->

4. Predefined replacement character strings can be specified for the 'prohibited' characters:

- "&lt;" instead of "<"
- "&gt;" instead of ">"
- "&amp;" instead of "&"
- "&apos;" instead of "'" (single quote)
- "&quot;" instead of """ (double quote)

3.7 Namespaces

To be able to use several XML documents from different name specifications in one XML document, you must ensure that element or attribute names are unambiguously assigned to one of the documents contained with its namespace. Thus a name can be unambiguously interpreted, even if it is defined in various specifications. The concept of the namespace was introduced to provide this level of unambiguity (see [NS specification]). These are the main points regarding the use of namespaces:

1. A namespace definition consists of a prefix and a URL, where the specification of the namespace is found (this is not analyzed by the UTM-XML API).
2. Every definition is assigned to an element node of the XML object.
3. In XML documents, namespaces are defined with the attribute xmlns="<url>" (Default-namespace) or xmlns:<prefix>="<url>". For example:
   ```
   <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd1="http://localhost:8080/wsdl/mynamespace.xsd"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap" >
   ```
   where the default namespace is defined with "xmlns".
4. The area of validity is determined as follows: the namespace is valid for the assigned element nodes and all element and attribute nodes within the relevant subtree, if no namespace is defined there with the same prefix.
5. If a prefix is attached to an element or attribute name, separated with ':' , the tag name is assigned to the appropriate namespace.
6. A namespace definition with an empty prefix is called a default namespace. With this type of definition, every element name (but not attribute) without a prefix in the area of validity is assigned to the default namespace.

The following features depend on the working procedures of the parser used:

1. If an XML object contains an element with a prefix which is not defined for this location, a dummy namespace is generated by the parser internally.
2. Dummy namespace definitions are recorded by the call KXLReadNSList but cannot be read with KXLSearchNS.
3. Attributes with undefined prefixes are not accepted. On converting or writing, a parser return code is given.

3.8 Structure of element names, attribute names and node lists

The name of an element is an arbitrary printable character string terminated by `\0`. It must conform to the constraints of the [XML specification] (see also section on Namespaces) and may additionally contain the characters "/", "[" and "]".
If a name contains a "/", this is interpreted as a separator for the structure components containing the data item. Thus, components of a structure can be passed by specifying "/": The name "Invoice/Address/Name" addresses an element "Name" in the structure "Address" within the structure "Invoice". The type and value passed in `t_value` refer to the element "Name". If no node "Invoice" of type `struct` exists at the time when the nodes are written, it is created implicitly (without content). If no node "Address" of type `struct` exists under the node "Invoice" (as a direct child node), this is likewise created implicitly (without content). Multiple elements in a structure are stored in the sequence in which they are created (no sorting).
If a node of type `struct` or `array` is created explicitly, additional information, such as the type name of the structure, can be stored as contents of the node.

If a name contains a "[", this is interpreted as an index operator in arrays. The name part after "[" should be a (printable) positive integer without leading zeros (index of the array element, beginning with 0, max. 98 digits long) and be terminated by "]". In this way individual array elements can be addressed. The (i+1)-th element in an `array` is written e.g. by specifying "Article[i]". If no node with the name "Article" and type

`array` has yet been created, it is created implicitly. The sorting of elements with not purely numeric indices or with leading zeros is described in the Appendix, in section 'Processing external elements'.
If the passed pointer points to the array node, only the index is specified as the name in the form "[i]" when an array element is accessed, if the (i+1)-th array element is to be written. If "[ ]", i.e. no index, is specified, then the first free array element is written. The elements in an array are arranged in ascending order by index.
If an array element is itself also a structure, the names of the structure elements are additionally separated from the index by "/", e.g.: Invoice/Article[2]/Order-No.
Always the relevant name part starting from the specified node must be specified. If, for example, an element with the name "Invoice/Article[2]/Order-No." is to be read, and the pointer to the node "Invoice/Article[2]" is specified in the call, then only "Order-No." is specified as the name.

If a name part includes a ':', the previous name part is interpreted as a namespace prefix (see the section on namespaces). Name parts without a prefix are assigned to the default namespace, if one has been defined. Name parts with different prefixes are recognized as different, even if the related URLs are the same. Indices of arrays cannot have a namespace prefix.

It is possible to have various elements with the same name under the same parent node. They are described in this API as a node list. To be able to access the individual elements of a node list, the one of the following expressions can be specified as a **node list position**: [+], [-], [++], [--] .
For a specified node, these expressions will look for the next ([+]), the previous ([-]), the last ([++]) or the first ([--]) of the sibling nodes with the same name. If a name begins with the node list position, the node list of the specified node is taken as the name.

If a complex element name is specified in a call, you can include a node list position as often as you wish. However do not use node lists one directly after another.
(Note: The same applies for the combination of array element specifications and node list positions – do no use several square brackets one after another).
The node list position always refers to the immediately preceding name part. As the last name part of complex names, only [+] is allowed in a write call, for read and position calls any position except [-] is allowed, where [--] has no meaning in this position.

The name and contents of attributes can be accessed in the same way as for elements. A particular attribute of an element is referred to by the element name, with '/@' and the attribute name added to show it is an attribute name. This means that the full name of the attribute 'purpose' of the element 'address' within the structure 'orders' is 'orders/address/@purpose' (e.g. with the content 'delivery' or 'invoice').
Names can be built in this way to write and read both attributes and elements. Attributes are always 'leaf nodes', so that only the last part of the name can begin with '@'. It is possible to position on attributes, but there is only use to do so when reading with local name (KXLFindNode).
If a name part contain a ‘@’, this is only recognized as an error on parsing (creating an object from the document).

3.9 Character sets/Encoding

XML objects are processed internally in UTF-8 (a variant of the Unicode character set) [UTF8], i.e. each document is converted into UTF-8 code when an XML object is built. If another encoding is used in the document, a valid encoding statement must be included in the document header. Without an encoding statement, the UTF-8 code will be taken as default.

When elements are written and read, names, contents, attributes, PIs etc are automatically converted from the character set which is used on the local computer (known as home encoding) to UTF-8 or back, so that the user interface does not have to perform any code conversion.
When a document is created from an XML object, this is converted from UTF-8 into the character set specified under encoding. If none is specified, the API assumes UTF-8 and does not convert it.
When an XML object is created, the character set specified in the home encoding is taken. This statement can be read with KXLGetDocEnc or changed with KXLSetDocEnc.

The character set used on the local computer is specified in the header file libxml/kxlinc.h under _KXL_HOME_ENC_STR (for printable names, see table) and _KXL_HOME_UTF8_STR (name in UTF-

8). The default is set as EBCDIC for BS2000 and as UTF-8 for all other platforms. This default can be read with **KXLGetHomeEnc.** With the first call of KXLInitEnv a user specific home encoding can be specified.
The most important code conversion routines are specified in the parser or in UTM XML. The following list shows the routines, and corresponding encoding statements.

Encoding name	Encoding
UTF8, UTF-8	UTF-8
UTF-16BE	UTF16BigEndian
UTF-16LE	UTF16LittleEndian
ASCII	ASCII
ISO-8859-1, …, ISO-8859-16	ISO-Latin-1, …, ISO-Latin-16
CP850	Codepage 850 (DOS-Latin-1)
EDF03DRV	BS2000 EBCDIC DF03 (German version) with Euro sign (0x9F)
EDF03IRV	BS2000 EBCDIC DF03 (International version) with Euro sign (0x9F)
EDF04DRV	BS2000 EBCDIC DF04 (German reference version)
EDF04_01	BS2000 EBCDIC DF041

Besides that alias names are predefined. So existing encoding handler functions can be addressed with other names. Their assignment can be read with the function **KXLGetEncodingAlias** and can be changed or deleted with the function **KXLSetEncodingAlias**. The following alias names are defined:

Alias name	Original Encoding name
EBCDIC	EDF04DRV
OSD_EBCDIC_DF04_DRV	EDF04DRV
OSD_EBCDIC_DF04_01	EDF04_01
EDF041	EDF04_01

If you want to work with other character sets, you can declare own conversion routines with the parser function xmlNewCharEncodingHandler**.** See [libxml internationalization support]. These calls are possible, for example, at the identified location in the KXLInitEncHdlr function. Besides that you can modify the existing EBCDIC encoding tables (EDFnnxxx, nn=03, 04) directly in the source and link the modified instead of the default tables to your application. See chapter 7 “Usage” and Appendix “Usage of user specific encoding functions”.

The following summary shows the transfer between different character sets. The following abbreviations are used:

H-Enc	= home encoding	= character set of the local platform
D-Enc	= document encoding	= character set of the current document
UTF-8	= internal encoding	= character set used internally by the parser

API – Input / - Output	UTM XML function	Parser
XMLdocument(D-Enc)	→ KXLConvDocToObj →	XML object (UTF-8)
XML document (D-Enc)	← KXLConvObjToDoc ←	XML object (UTF-8)
root name /-content (H-Enc)	→ KXLCreateNewObj →	XML object (UTF-8)
element name/- content (H-Enc)	→ KXLWrite →	node in the XML object (UTF-8)
element name (H-Enc)	→ KXLRead →	node in the XML object (UTF-8)
element content (H-Enc)	← (return)	←
element name/-content (H-Enc)	← KXLReadXxx ←	node in the XML object (UTF-8)
element-content (H-Enc)	← (return)	←
element name (H-Enc)	→ KXLSetSubObject →	node in the XML object (UTF-8)

Normally, H-Enc = D-Enc (e.g. = EBCDIC in BS2000), but EBCDIC documents can also be edited on Windows or UNIX platforms.

Care must be taken to ensure that the usual transfer tools automatically convert a text document. This means, for example, that a document transferred from BS2000 to Windows via FTP containing the string "encoding='EBCDIC'", is available on Windows in ISO-8859-1.

For handling an XML document on various platforms it is useful to create it with the encoding attribute encoding="UTF-8". Then you can transfer the document without conversion. Another possibility is described in Appendix, section “Handling of XML documents without encoding attribute”.

Example of the declaration of conversion routines to a private character set:

```
#include <libxml/encoding.h>
/* input for parser function in UTF-8! */
const static char UTF8_EDF031string[7]=  {69,68,70,48,51,49,0};
/* ASCII for "EDF031\0" */
/* conversion routines EDF031ToUTF8 and UTF8ToEDF031 must be written by user
*/
int     EDF031ToUTF8 (unsigned char* out, int *outlen, const unsigned char*
in, int *inlen);
int     UTF8ToEDF031 (unsigned char* out, int *outlen, const unsigned char*
in, int *inlen);
xmlCharEncodingHandlerPtr EncHdlr;

EncHdlr = xmlNewCharEncodingHandler ( UTF8_EDF031string, EDF031ToUTF8,
UTF8ToEDF031);
if ( EncHdlr == NULL )
{ /* declaration failure -> error handling */ }
```

3.10 XML schema validation

XML schema is recommended by the W3C to define XML document structures. (see [XML schema specification]) In contrast to classic XML DTDs, the structures are described in the form of an XML document. Furthermore, a large number of data types are supported. When an XML schema-based XML document is parsed, the validity of the structure and proper usage of types is always checked as well. [XML schema – Definition]

With the schema validation functions (see chapter XML API for C/C++, section XML schema validation), the UTM XML API supports the parsing of XML schemas and the validation of XML documents against an XML schema.

3.11 File handling

The schema validation functions not only access data in memory, but also access files. A file can also be specified as a parameter, for example, when parsing a schema. In addition, parts of the schema may be stored in other files that are referenced using include, import, or other elements of the schema. The files are identified by their URL. To obtain a consistent data access design on the various platforms and to permit data access at all in the first place, two handlers that control file access are declared in UTM XML using parser functions. Furthermore, it is possible for the user himself to declare such handler routines.

Caution: When you use user-defined handler routines, you might interfere with the operation of the UTM XML functions described here. This means that the file handling may not be performed as described in the interface functions.

3.11.1 Entities Loader

During the initialization of UTM-XML (KXLInitEnv function), an interface-specific handler routine named KXLResolveEntity is declared that contains the following functionality:

The general structure of a URL of type http or ftp is as follows:

<URL type>://[<user>[:<password>]@]<server>[:<port>]/<path>?<request>#<fragment>

where the specification of file names in the last part (?<request>#<fragment>) is ignored and <path> can be specified in the form <directory>/<filename>.

In the KXLConvDocToObjAndValid , KXLParseSchema, and KXLParseSchemaFile functions, you can specify a local file directory in the pDir parameter so that files referenced in XML or schema documents are read from this local file directory instead of the network.

In BS2000 you can specify a user ID (with '$', without '.', e.g. "$UTMXML") in pDir. If the files are to be referenced from the current user ID, then specify an empty string or a string containing only spaces in pDir.

In this case, a URL with the UTM XML-specific Entity Resolver is processed as follows:

If <URL type> = "http" or "ftp", then the part

<URL type>://[<user>[:<password>]@]<server>[:<port>]/<directory>

of the name is replaced by the local directory pDir=<mydir> and terminated by the operating system-specific file separator character (/ or \ or .) (or an empty string when <mydir> is an empty string).

If <URL type> = "file", then the "file://" part of the name at the beginning is deleted. The local directory specification is ignored. All other file names are not converted.

Example: "http://www.w3.org/2001/XMLSchema" is converted to "<mydir>/XMLSchema". If the files are to be read from the local directory, then <mydir> = "" is to be specified, i.e. the URL is converted to "XMLSchema".

It is possible for users to incorporate their own routine written according to the description of the parser [IO Interfaces] instead of the handler mentioned above. In this case note the following:

- With the parser function xmlSetExternalEntityLoader you can declare your own ResolveEntity handler routine that replaces a URL by a user-specific file name. A user-specific ResolveEntity handler must be declared after the initialization of the UTM XML interface (KXLInitEnv).
- If you declare your own ResolveEntity handler to the parser, the reading of the files from the local directory specified in the `pDir` parameter as described above is disabled.
- You can reset the ResolveEntity handler back to the default handler of UTM XML by calling the KXLInitEnv function again after the declaration of your own ResolveEntity handler.

3.11.2 IO handler in BS2000

Since the parser functions in BS2000 work internally with ASCII strings, it is necessary to declare a separate UTM XML IO handler for all other functions expecting EBCDIC strings such as file handling calls (open, read), for example. It consists of four functions:

- of type xmlInputMatchCallback that checks if the IO handler is to process this file
- of type xmlInputOpenCallback that opens the file with the specified file name
- of type xmlInputReadCallback that reads the file, and
- of type xmlInputCloseCallback that closes the file.

These functions are called by the parser when a file is to be read in.

During the initialization of UTM-XML (KXLInitEnv function), the interface-specific IO handler is declared with the functions KXLIOHandler<func> with <func> = Match/Open/Read/Close offering the following functionality:

If the file name passed starts with "file://" or does not contain any URL type specifications, then the Match routine returns 1 (i.e. the file will be handled). The Open/Read/Close functions open, read, and close the file taking the encoding problem into account in BS2000.

It is possible for users to incorporate their own routines (add new ones or replace existing ones) for the handlers mentioned above at their own risk. They have to be written according to the description of the parser. Note the following in this case:

- If you declare your own IO handler in BS2000 to the parser (with xmlRegisterInputCallbacks), the order of the declarations is important because while processing a URL, all IO handler functions of type xmlInputMatchCallback are called until the first positive return. This means that a user-specific IO handler must be declared before the initialization of the UTM XML IO handler.
- You can reset the IO handler to the default handler of UTM XML by calling the KXLInitEnv function again after the declaration of your own IO handler.
- Caution: The parser creates a table of the declared IO handlers that is processed recursively. This means that if an IO handler rejects handling (0 is returned by the IO handler Match routine), then the previously declared IO handler is called. The size of this table is limited, i.e. a maximum of 15 declaration calls is possible.
- The function xmlRegisterInputCallback and the function types xmlInput<func>Callback with <func> = Match/Open/Read/Close are declared in the parser header file xmlIO.h.

3.12 Notes

1. All (external) functions and procedures begin with the prefix KXL.
2. All functions beginning with the prefix xml are parser functions. They are only available in C and are not described further in this document. Parser functions require input in UTF-8 code, also in BS2000.
3. In BS2000 the length limit on external names of 7 or 8 characters no longer applies. Correspondingly, C program units, which call the functions of this API, must be compiled and linked as LLMs.

4 XML API for C / C++

The API is subdivided into nine groups, i.e. into calls

1. to convert the data types into the t_value structure and back,
2. to create an XML object,
3. to navigate in the XML object,
4. to read an XML object and
5. to convert from XML object to XML document and vice versa, and to release XML documents.
6. to handle character sets
7. to manage namespaces
8. to validate against an XML schema files and
9. to initialize and perform diagnostics.

To call the interface, the header `libxml/kxlinc.h` must be specified in the program. This contains all the necessary definitions and function prototypes.

The return code is optional in all functions in which it is defined. In other words, the NULL pointer can be passed instead of the pointer to a return code. Then, no return code is returned.

4.1 Initialization of the UTM XML interface

Function prototype:

```
void     KXLInitEnv(char * encFuncName,
                    xmlCharEncodingInputFunc encFuncToUTF8,
                    xmlCharEncodingOutputFunc encFuncFromUTF8,
                    short* pRetCode);
```

Parameter:

Type	Name	Comment
char *	encFuncName	In: Name of the home encoding
xmlCharEncodingInputFunc	encFuncToUTF8	In: Encoding function to UTF-8
xmlCharEncodingOutputFunc	encFuncFromUTF8	In: Encoding function from UTF-8
short*	pRetCode	Out:Return code unless NULL was specified

To set up the environment for the UTM XML interface, various functions of the parser for UTM XML and the interface-specific trace environment must be initialized. Calling this function initializes the trace environment (KXLTSENV call) and declares various parser-specific handlers (encoding handler, entity resolver, IO handler (BS2000 only)) and a user specific home encoding if wanted. Furthermore, version and consistency checks are performed.

If encFuncName is specified a user specific home encoding will be defined. For an encoding that is already declared in the UTM-XML interface (see section “Character sets/Encoding) NULL can be specified for the functions encFuncToUTF8 and encFuncFromUTF8.Then the predefined encoding functions are used. If the encoding isn’t declared or if user encoding functions should be used, you have to specify the addresses of two functions of the following prototype:

```
int homeEncodingToUTF8 (unsigned char* out, int *outlen,
                        const unsigned char* in, int *inlen);
int UTF8ToHomeEncoding (unsigned char* out, int *outlen,
                        const unsigned char* in, int *inlen);
```

The meaning of the parameters are as follows:

out	Address of the output buffer where the converted string is written to
outlen	in: length of the output buffer, out: length of the converted output string
in	Address of the input buffer containing the string to be converted
inlen	in: length of the input buffer (number of bytes), out: number of the characters consumed

Return value (int) = n, length of the converted output string,
-2, if an error was detected during conversion,
-1, other errors.

If encFuncName isn't specified (NULL) no user specific home encoding will be defined.

Notes:

1. This function only need to be called explicitly when functions of the parser that use the handler mentioned above are called directly before the UTM XML functions are called or if a user specific home encoding should be defined. Then this function must be called in each task or process as the first call of the UTM-XML interface. When using the UTM-XML interface within a UTM application this call must be given in each task/process, e.g. in the start exit.
2. The first time a UTM XML function is called that returns xmlNodePtr, xmlNsPtr, or xmlSchemaPtr or a function for character conversion is called (i.e. in KXLCreateNewObj, KXLConvDocToObj, KXLConvDocToObjAndValid, KXLParseSchema, KXLParseSchemaFile, KXLStringToUTF8, KXLStringFromUTF8, KXLGetEncodingAlias, KXLSetEncodingAlias, not for the functions converting the t_value structures, nor in the KXLGetHomeEnc and KXLTSENV calls), the KXLInitEnv function is called implicitly. In this case no home encoding is specified (KXLInitEnv (NULL, NULL, NULL, &retCode)). The corresponding return code is returned in case of an error.
3. It is also possible for users to incorporate their own routines written according to the description of the parser for the handler mentioned above. (see section 3.11 File handling) If you call this function explicitly again after that, then the EntityResolver and IO handler (in BS2000) are reset to the UTM XML default routines. (see also section 3.11 File handling)
4. The consistency checks and the definition of the home encoding are only processed on the first call of this function (in each task/process).

Return codes:

Name (Value)	Error Type	Comment -> Action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_VERSION_ERR (23)	U	The versions of the UTM XML components are inconsistent
KXL_RC_PARSER_VERS_NOT_SUPP (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate newer parser version
KXL_RC_UNKNOWN_ENCODING (30)	U	Encoding name is unknown to the parser -> declare the conversion routines for encoding in the parser
KXL_RC_ENCODING_CHANGE_ERR (36)	U/S	No encoding handler could be set up for the specified home encoding
KXL_RC_IO_HNDLR_INIT_ERR (45)	U	To many IO handlers defined

where
I= Information, U=User error, S=System error

4.2 Type conversion functions

Conversion functions are provided for certain data types to make the interface easier to handle for the user. Because the conversion takes place before the writing and after the reading interface call, the user has the opportunity to write other conversion routines, including for his own data types, and to transfer this data at the interface (see also section "Processing external documents").

The C data types `short, int, long, float, double and char` are referred to below as **simple data types**, and the data types `string` (character string), `struct` (structure) and `array` as **complex data types**.

4.2.1 Converting to the `t_value` structure

Simple data types

Function prototypes:

```
t_value KXLFromShort(short s);
t_value KXLFromInt(int i);
t_value KXLFromLong(long l);
t_value KXLFromFloat(float f);
t_value KXLFromDouble(double d);
t_value KXLFromChar(char c);
```

Parameters:

Type	Name	Comment	
short	s	In:	short value to be converted
int	i	In:	int value to be converted
long	l	In:	long value to be converted
float	f	In:	float value to be converted
double	d	In:	double value to be converted
char	c	In:	char value to be converted

Function result:

Type	Comment
t_value	structure with the converted value

The `short, int, long, float, double` or `char` value to be converted must be specified in each case as the parameter in the conversion routines for the simple data types `short, int, long, float, double` and `char`.

As return value, the functions supply a structure of type `t_value`, consisting of two pointers to character strings (`char*`). The first string contains a printable "short", "int", "long", "float", "double", "char" depending on type, the second the printable value. The storage area for these strings is managed by the conversion functions. A new call to a conversion function overwrites the result of the preceding call. In other words, the caller must copy the result into his own storage areas if necessary.

Example:
```
 t_value      tval;
 double       Summe = 99.90;
 int          i;
 char         tval_Type[11];
 char         tval_Value[201];
 tval = KXLFromDouble (Summe);
 i = MIN (strlen(tval.pType),10);
 strncpy (tval_Type, tval.pType, i );
 tval_Type[i] = '\0';
 i = MIN (strlen(tval.pValue), 200);
 strncpy (tval_Value, tval.pValue,i);
 tval_Value[i] = '\0';
```

Result: tval_Type = "double"
tval_Value = "99.90"

Complex data types

Function prototypes:
```
t_value KXLFromString(char* s);
t_value KXLFromStruct(char* s);
t_value KXLFromArray(void);
```

Parameters:

Type	Name	Comment
`char *`	`s`	`In:   Value to be converted`

Function result:

Type	Comment
`t_value`	`Structure with the converted value`

Conversion functions are also provided for the complex data types `string` (`\0` terminated character string), `struct` (structure) and `array` (vector). The conversion for `struct` and `array` is only required when an XML subobject of type `struct` or `array` is to be created explicitly (see section Program interface..., Structure of the element names). A character string can be passed as value for type `struct`. Thus, for example, the type name of the structure can be entered, although the value parameter can also be supplied with NULL. The conversion function for `array` has no value parameter (void).

The parameter of the function is the `char*` value to be converted. As return value, the functions supply a structure of type `t_value`, which consists of two pointers to character strings (`char*`). The first string contains a printable "string", "struct" or "array" depending on type, while the second pointer contains the original address of the input value (no copy).
Warning: The routines do not check the correctness of the value to be converted!

Example:
```
t_value   tval;
char *    structtyp = "Addresses";
tval = KXLFromStruct (structtyp);
Result:   tval.pType      ->"struct"
          tval.pValue     ->"Addresses"
```

4.2.2 Converting from the `t_value` structure

When an object is read, values of type `t_value` are returned at the call interface in a similar way to when an object is written. These values can then be converted again by the caller via conversion routines. The inverse type conversion functions are provided for the same simple data types, and in addition for the data types `string` and `struct`.

Simple data types

Function prototypes:
```
short   KXLToShort  (t_value tval, short* pRetCode);
int     KXLToInt    (t_value tval, short* pRetCode);
long    KXLToLong   (t_value tval, short* pRetCode);
float   KXLToFloat  (t_value tval, short* pRetCode);
double  KXLToDouble (t_value tval, short* pRetCode);
char    KXLToChar   (t_value tval, short* pRetCode);
```

Function parameters:

Type	Name	Comment
t_value	tval	In: Value to be converted
short*	pRetCode	Out: Return code, unless NULL was specified

Function result:

Type	Comment
short	converted short value
int	converted int value
long	converted long value
float	converted float value
double	converted double value
char	converted char value

The functions convert the values in `t_value` into the desired types, provided this is possible. Where source and target type are different, a return code is set, but the function is executed, with the value possibly being truncated when converted (e.g. with float -> int). If one of the pointers of t_value is not set (NULL), 0 or space (as char) is returned.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_DIFF_TYPES(2)	I	Type in t_value and requested type different. The result value may be truncated.
KXL_RC_INVALID_T_VALUE (17)	U	One of the pointers specified in t_value is NULL ->correct value

where
I= Information, U=User error, S=System error

Example:
```
t_value tval;
char * T_type = "float";
char * T_value = "15.90";
float f;
tval.pType = T_type;
tval.pValue = T_value;
short Retcode;
f = KXLToFloat (tval, &Retcode);
```

Complex data types

Function prototypes:
```
char*  KXLToString(t_value tval, short* pRetCode);
char*  KXLToStruct(t_value tval, short* pRetCode);
```

Function parameters:

Type	Name	Comment
t_value	tval	In: Value to be converted
short*	pRetCode	Out: Return code, unless NULL was specified

Function result:

Type	Comment
char *	converted string

The functions convert the values in `t_value` into the desired types. Where source and target type are different, a return code is set. This also applies if an empty string was specified for `t_value.pType`. The pointer `t_value.pValue` is returned as the return value.
The function KXLToStruct is only necessary if the information stored in a struct node (see above, e.g. type name of the structure) is to be accessed.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_INVALID_T_VALUE (17)	U	One of the pointers specified in t_value is NULL -> correct value

where
I= Information, U=User error, S=System error

Example:
```
 t_value tval;
 char * T_type = "struct";
 char * T_value = "Adressen";
 char * structtyp;
 tval.pType = T_type;
 tval.pValue = T_value;
 short Retcode;
 structtyp= KXLToStruct (tval, &Retcode);
```

4.3 Write functions

4.3.1 Creating an XML object

Function prototype:

```
xmlNodePtr    KXLCreateNewObj(char* pName,t_value tval,short* pRetCode);
```

Parameters:

Type	Name	Comment
`char*`	`pName`	`In:   Name of the root element`
`t_value`	`tval`	`In:   t_value of the root element`
`short*`	`pRetCode`	`Out:  Return code, unless NULL was specified`

Function result:

Type	Comment
`xmlNodePtr`	`Pointer to the root node of an empty XML object`

The function generates the 'header' of an XML document and creates the root node with the specified name `pName` and content `tval`. If `pName` is the NULL pointer, or if it points to the empty string, the root node with the name UTM-XML and version attribute is built. If the pointers `pType` and `pValue` are NULL, or if they point to the empty string ("\0"), no type attribute is created or no element content stored.
The return value of the function is a pointer to the root node of an empty XML object. If desired, the function returns a return code.

When a new XML object is created, the encoding statement defined under _KXL_HOME_ENC_STR is set as the encoding value (see the section on character sets). This value can be read and changed with **KXLGetDocEnc** or **KXLSetDocEnc.**

When the first call is issued, the KXLInitEnv function may be called implicitly (see section 4.1 Initialization of the UTM XML interface). The corresponding return code is returned and no object is created in case of an error.
If no XML document could be created, then the corresponding return code is returned (KXL_RC_NO_ROOT_CREATED(4)). You can request more detailed information on the error via KXLGetLastParserError or by looking in the trace file.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_ROOT_CREATED (4)	S	The parser was not able to create a new object; -> see parser message. (Trace file or KXLGetLastParserError)
KXL_RC_INVALID_NAME (19)	U	When the element name was analyzed, a syntax error in the use of "[", "]" "/" or @ was found -> state a valid name
KXL_RC_VERSION_ERR (23)	U	Inconsistent versions of UTM-XML components
KXL_RC_PARSER_VERS_NOT_SUP P (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate newer parser version
KXL_RC_UNKNOWN_ENCODING (30)	U	The encoding name is unknown to the parser -> declare the conversion routines for encoding in the parser
KXL_RC_ENCODING_CHANGE_ERR (36)	U/S	No encoding handler could be set up for the specified home encoding

where
I= Information, U=User error, S=System error

Example:

```
t_value     tval = {NULL, NULL};
short       Retcode;
xmlNodePtr          pRoot;
pRoot = KXLCreateNewObj ("mydoc", tval, &Retcode);
```

creates the root node <mydoc/>

4.3.2 Writing an element

Function prototype:

```
xmlNodePtr   KXLWrite(xmlNodePtr pNode,char* pName,t_value tval,
                          short* pRetCode)
```

Parameters:

```
Type          Name        Comment
xmlNodePtr    pNode       In:   Pointer to a node of the XML object.
char*         pName       In:   Name of the element or attribute
t_value       tval        In:   t_value of the element or value of the
                                      attribute.
     short*         pRetCode     Out:  Return code, unless NULL was specified
```

Function result:

```
Type          Comment
xmlNodePtr    Pointer to the written node
```

When the function is called, the pointer to a node of the XML object, the name of the element and the structure `t_value` containing the type and the value of the element must be specified. The pointer to the written node (or, in the event of an error, NULL or the pointer to the node in which the error occurred) and the return code are passed to the caller as return value.

The specified pointer can be any node of the XML object. As the name parameter always the relevant name part starting from the specified node must be specified (see chapter Program interface UTM XML, section Structure of element names…). Names must conform to the constraints of the [XML specification]; names can also contain the structure separators "/", "[" and "]". With an empty name or name parts (index in [ ] or between two /), a return code is returned. detects Other syntactical errors are detected by the parser when KXLConvDocToObj is called.

If there is no name, the element indicated by the pointer (pNode) is edited. It is only possible to overwrite node list elements if the pointer indicates the appropriate element and the empty name is given.

In structures, a type name, for example, can be passed as value; with type array, the pValue pointer must have the value NULL.
If pType points to the empty string, no type attribute is generated for the desired element and all implicitly created elements of type "struct". type="struct" (interior nodes) or type="string" (leaf nodes) is assumed implicitly for all nodes without type attribute.

When an array element is written, no check is made to verify whether all elements of the array are of the same type.

If an error occurs, all elements generated implicitly up to that point are retained.

If the component with the specified name and type is not yet present, it is appended as a new node after the last sibling node. Array elements are sorted in ascending order by index.
If the component with the specified name and type is already present in the specified (sub)object, the existing content is replaced by the specified value. If the NULL pointer is passed as the value, the existing content is deleted.
If the component with the specified name but of different type is already present, the write operation is rejected with return code KXL_RC_TYPE_MISMATCH. In this case, type = 'string' or 'struct' and the empty type attribute are considered different. If the component is to be overwritten nonetheless, it must first be read with the delete flag set (and hence deleted). It can then be written as desired.
Warning! This also applies to already existing subtrees: If the value of an interior node (e.g. value = typedef of a struct node) is overwritten, the entire subtree (i.e. the structure) is also overwritten, i.e. deleted.

If the specified element name ends with '[+]', the preceding name part is interpreted as a node list name (see the section on "Building element and attribute names and node lists") and created as a new element in the node list. It is not sorted, and it is not checked if an element of this node list exists or if the type statements match. If the name ends with [++], [-] or [--], the call is rejected with a return code.
If the name is only given as "[+]", a new element is set up in the node list, with the name of the currently specified node. If a name part (not at the start or end) contains a node list position, when there is a position call, it will be positioned on the next ([+]), last ([++]) or first ([--]).[-] is rejected with a return code.
If a name begins with a node list position, the position refers to the node list, to which the specified node belongs. If [+] is entered as one of the internal name parts, and no other element of the specified node list is present, it is implicitly created. If no element is found in the other node list positions, a return code is given. If the specified name has square brackets with content other than that described above, it is interpreted as the index of an array.

If an attribute name is entered as a name, i.e. the last name part begins with '@', an attribute with the value indicated by t_value.pValue is created for the higher-level element. If an attribute of the name is present it is overwritten or deleted, if the zero counter has been passed. The value of t_value.pType is ignored. If a name part other than the last one starts with '@' writing is rejected with the return code KXL_RC_INVALID_NAME

Every individual name part can contain a namespace prefix in the format "prefix:name". A name is then only the same if the prefix is the same (see section Namespaces). The prefix internally assigns the name to namespace definition. Names without a prefix are assigned to the default namespace. If no valid namespace definition is found for an element, a dummy definition is created.
For example: in the name "student/ns1:subject/ns2:grade", "student" belongs to the default namespace, "subject" to the namespace ns1 and "grade" to the namespace ns2.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_NODE_FOUND (1)	I	Node with the specified name not found
KXL_RC_ALLOC_ERROR (11)	S	No additional storage space could be requested with malloc/realloc
KXL_RC_TYPE_MISMATCH(12)	U	When a node is to be overwritten, the existing type does not match the specified type -> if you want to overwrite the node, first delete it, then rewrite
KXL_RC_NO_CHILD_CREATED (13)	S/U	Error when creating a new node -> analyze the parser message
KXL_RC_NO_TYPE_ATTR_FOUND(14)	S/U	While overwriting a node, no type attribute was found; no overwriting is possible.
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_EMPTY_NAME(18)	U	Name was empty or when the element name was analyzed, an empty name part was found; writing was aborted -> specify valid name.
KXL_RC_INVALID_NAME (19)	U	When the element name was analyzed, a syntax error was detected in the use of "[", "]" , "/" and @ -> specify valid name
KXL_RC_NAMESPACE_WRITE_ERROR (33)	S/U	The parser could not write the namespace definition correctly, see parser error messages

where
I= Information, U=User error, S=System error

Example:

```
float Price=1.75;
KXLWrite(pNode, "Price", KXLFromfloat(Price), NULL);
```

4.4 Conversion functions

4.4.1 Converting an XML object into an XML document

Function prototype:

```
char *  KXLConvObjToDoc    (xmlNodePtr pNode, char* pStylesheet,
                            int* pBufLen, short* pRetCode);
```

Parameters:

```
Type          Name         Comment
xmlNodePtr    pNode        In:   Pointer to the root node of the
                                        XML object.
char*         pStylesheet  In:   Content of the stylesheet PI
int*          pBufLen      Out:  Length of the buffer, unless NULL was
                                 specified
short*        pRetCode     Out:  Return code, unless NULL was specified
```

Function result:

```
Type    Comment
char*   Buffer containing the XML document
```

In order to create an XML document from an XML object, the function KXLConvObjToDoc must be called. This function expects the pointer to the root node of the XML object as input parameter. The document is converted from the XML object in UTF-8 into the character set specified under encoding. If no encoding is specified, the document remains in UTF-8 encoding. The encoding statement can be created or changed with KXLSetDocEnc.
If no conversion routine has been declared for the specified character set (see the section on character sets), no document is produced and a corresponding return code is given.

The function converts the XML object into an XML document and returns the address of the buffer which contains the XML document and is terminated by \0 (in the case of an error, the NULL pointer). If desired (pBufLen not NULL), the length of the buffer is also returned. A stylesheet PI with the specified content (in the format <?xml-stylesheet ... ?>) is inserted after the 'header' of the document (<?xml...?>) if pStylesheet is not NULL or points to the empty string. If a stylesheet PI already exists it is overwritten with the new PI. The storage area of the buffer is overwritten with the next call to KXLConvObjToDoc.

Note:

- Output formatting with additional line feeds is done only for elements, which only have element child nodes. For other elements, which have text or entity child nodes, no additional line feed will be inserted, because the content of the node would be corrupted.
- It is also possible to create an XML document without encoding attribute in an encoding different from UTF-8. See Appendix, section “Handling of XML documents without encoding attribute”.
- If the XML document could not be created, then the corresponding return code is returned (KXL_RC_NO_CONVERSION_TO_DOC(7)). You can request more detailed information on the error via KXLGetLastParserError or by looking in the trace file.
- When the root node of a schema object is specified, a schema document can also be created, of course. You get the root node of a schema C via the pointer to the schema structure in the following manner:
  ```
      xmlSchemaPtr->doc->children
  ```
  The KXLSchemaGetRoot function is available for this purpose at the Cobol interface.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_CONVERSION_TO_DOC (7)	S	Error during creation of the document ->see message of the parser. (Trace file or KXLGetLastParserError)
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object; Possible cause: object destroyed or created incorrectly -> generate error documentation
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_CONVERSION_ERROR (22)	S/U	Error in the code conversion.

where
I= Information, U=User error, S=System error

4.4.2 Converting an XML document into an XML object

Function prototype:
```
xmlNodePtr    KXLConvDocToObj(char* pBuffer, short* pRetCode);
```

Parameters:
```
Type    Name         Comment
char*   pBuffer      In:   Buffer containing the XML document
short*  pRetCode     Out:  Return code, unless NULL was specified
```

Function result:
```
Type          Comment
xmlNodePtr    Pointer to the root node of the XML object
```

Before it can be processed, an XML document must be converted into an XML object. To this end, the function KXLConvDocToObj is called, with the buffer containing the complete (\0-terminated!) XML document as input parameter. The result of the function is the pointer to the root node of the XML object. If an error occurs, NULL is returned.

If a document to be converted contains an encoding statement, this is placed in the object tree and all names and content are converted to UTF-8 and placed in the object tree.
If there is no encoding statement UTF-8 is assumed. It is also possible to process an XML document without encoding attribute in an encoding different from UTF-8. See Appendix, section “Handling of XML documents without encoding attribute”.
A code conversion routine (to and from UTF-8) must be declared for the encoding specified in the encoding statement, otherwise an error will be returned and no object will be created (see the section on character sets).

When the first call is issued, the KXLInitEnv function may be called implicitly (see section 4.1 Initialization of the UTM XML interface). The corresponding return code is returned and no object is created in case of an error.
If the XML document is not well-formed, then the corresponding return code is returned (KXL_RC_PARSER_ERROR(20)). You can request more detailed information on the error via KXLGetLastParserError or by looking in the trace file.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_MALLOC_FOR_ PARSER_1(5)	S	Not enough storage space available for the object -> increase size of program storage space
KXL_RC_NO_MALLOC_FOR_ PARSER_2(6)	S	Not enough storage space available for the object -> increase size of program storage space
KXL_RC_NO_CODE_ CONVERSION (8)	U	Beginning of the document does not match the XML start tag '<?xml' or '<?XML' in EBCDIC or ASCII -> correct document or perform code conversion
KXL_RC_EMPTY_DOC (9)	U	Converted object contains no node
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_PARSER_ERROR (20)	S/U	Error found in the parser. The message or parser RC can be recovered with KXLGetLastParserError
KXL_RC_NO_CONTEXT_FOR_ PARSER (21)	S	Error in setting up an internal management area in the parser -> maybe deal with memory bottleneck
KXL_RC_CONVERSION_ ERROR (22)	S/U	Error in code conversion
KXL_RC_VERSION_ERR (23)	U	Inconsistent UTM XML module versions
KXL_RC_PARSER_VERS_NOT_SUPP (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate newer parser version
KXL_RC_UNKNOWN_ ENCODING (30)	U	The encoding name is unknown to the parser -> declare the conversion routines for encoding in the parser
KXL_RC_ENCODING_CHANGE_ERR (36)	U/S	No encoding handler could be set up for the specified home encoding

where
I= Information, U=User error, S=System error

4.4.3 Releasing an XML object storage space

Function prototype:
```
void   KXLFreeObj(xmlNodePtr pNode, short* pRetCode);
```

Parameters:
```
Type       Name      Comment
xmlNodePtr  pNode    In:   Pointer to the root node of the XML object
short*     pRetCode  Out:  Return code, unless NULL was specified
```

This call releases the storage space needed for the XML object. It should always be executed when the XML object concerned is no longer being processed.
After this, addresses of the released object may no longer be accessed.
Caution: The XML object of a schema should not be freed with KXLFreeObj but should be freed together with the entire XML schema structure by calling KXLFreeSchema.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object. Possible cause: object destroyed or created incorrectly -> generate error documentation
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer

where
I= Information, U=User error, S=System error

4.5 Navigating in the XML object

4.5.1 Positioning on an XML subobject

Function prototype:

```
    xmlNodePtr     KXLSetSubObject(xmlNodePtr pNode, char* pName,
                   short* pRetCode);
```

Parameters:

Type	Name	Comment
xmlNodePtr	pNode	In: Pointer to the node of the XML object where the positioning is started
char*	Name	In: Name of the element onto which the pointer is to be positioned
short*	pRetCode	Out: Return code, unless NULL was specified

Function result:

Type	Comment
xmlNodePtr	Pointer to the searched-for node of the XML object

In both writing and reading, it can be useful to restrict access to a subtree in order to avoid having to specify the fully qualified name every time. This increases performance, since it is not necessary to perform the search across all levels of the object tree when a node is accessed. For this reason this function provides a positioning on a subtree of the XML object. It returns the pointer to a node which forms the root of a subtree in the XML object, in accordance with the specified structure or array. Via this pointer, partially qualified names can be used with the access functions (e.g. KXLWrite/KXLRead) to access the components of the structure or array.
If empty name parts are entered (index in [] or between two /) a return code is given.
If an error occurs, the NULL pointer is returned.

If the name contains one of the node list positions [--], [+] or [++], the pointer can be positioned on the first, second or the last node list element. The [-] can only be specified at the start of a name, and in this case the pointer is placed on the node list element preceding the specified node. If [-] is found elsewhere, the call is rejected with a return code. With [+] at the start of a name, the pointer is positioned on the node list element following the specified node. You can position the pointer on other elements of the node list, or read them, with the returned pointer.
If the square brackets contain other characters, these are interpreted as the index of an array.

If you are working with namespaces, you must specify the name parts with the corresponding prefixes.

Note: The pointer can be positioned on attributes, but there is no special reason for doing this. For information on building names, see section "Building element and attribute names and node lists".

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_NODE_FOUND (1)	I	Node with the specified name not found
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_INVALID_NAME (19)	U	An error in syntax was found in the use of "[", "]" "/" or @ when an element name was analyzed. -> state a valid name.
KXL_RC_CONVERSION_ ERROR (22)	S/U	Code conversion error
KXL_RC_NODE_TYPE_NOT_ ALLOWED (28)	U	Only an element node or attribute node is allowed with this call.

where
I= Information, U=User error, S=System error

4.5.2 Positioning onto the root node of the XML object

Function prototype:
```
xmlNodePtr    KXLSetRootNode(xmlNodePtr pNode, short* pRetCode);
```

Parameters:
```
Type          Name        Comment
xmlNodePtr    pNode       In:   Pointer to a node of the XML object.
short*        pRetCode    Out:  Return code, unless NULL was specified
```

Function result:
```
Type          Comment
xmlNodePtr    Pointer to the root node of the XML object
```

In some cases it is necessary to pass from working in a subtree back to the entire XML object, whether because an access at some other point is necessary, or because the work on the XML object has been completed and it is to be converted and sent.
Any node of the XML object can be passed as the parameter. The return value is the root node of the XML object or, if an error occurs, the NULL pointer.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object Possible cause: object destroyed or incorrectly created -> generate error documentation
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer

where
I= Information, U=User error, S=System error

Example for recognizing root nodes:
```
if (KXLSetRootNode(pNode, NULL) == pNode)
{ /* root reached, stop working */ }
```

4.5.3 Positioning onto the next-higher XML subobject

Function prototype:

```
xmlNodePtr    KXLSetParentNode(xmlNodePtr pNode, short* pRetCode);
```

Parameters:

```
Type          Name         Comment
xmlNodePtr    pNode        In:   Pointer to a node of the XML object.
short*        pRetCode     Out:  Return code, unless NULL was specified
```

Function result:

```
Type          Comment
xmlNodePtr    Pointer to the next-higher node of the XML object
```

In some cases it is useful to position from a subtree back to the next-higher subtree in order, for example, to access the structure that follows the current structure.
Any node of the XML object can be passed as parameter. The next-higher node within the XML object is returned. If the root node is specified, NULL is returned.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer

where
I= Information, U=User error, S=System error

4.6 Read functions

In an existing XML object (newly created or converted from an XML document), the individual elements of the object can be read using read functions.

4.6.1 Reading using names

Function prototype:

```
t_value KXLRead(xmlNodePtr pNode, char* pName, int delete,
                          short* pRetCode);
```

Parameters:

Type	Name	Comment	
xmlNodePtr	pNode	In:	Pointer to the node of the XML object.
char*	pName	In:	Name of the element
int	Delete	In:	KXL_TRUE / KXL_FALSE
short*	pRetCode	Out:	Return code, unless NULL was specified

Function result:

Type	Comment
t_value	t_value of the element

The function KXLRead can be used to address an element via its name. KXLRead returns the type and value of the data item in the structure of type `t_value`, which can be converted using the functions described in section "Type conversion functions". If no type attribute is present, the empty string is returned as type. For interior nodes, the type corresponds to type="struct"; for leaf nodes, type="string".
As with writing, reading can be performed in two ways: If the root node is specified, a data item can be addressed via the fully qualified name. But reading can also be performed at multiple levels: KXLSetSubObject can be used to position in the XML object. Using the pointer to a subobject obtained in this way, KXLRead and the component name or array index (in the form [i]) can be used to access the individual data.
If an empty name is specified (spaces), the type and value of the specified node (pNode) are returned. This is only possible for element nodes and not attribute nodes. These can be read directly through the call KXLReadNode.

The first element of a node list will be read as an individual element through a node and the statement of the relevant names. If a node list position is specified ([+],[-],[++] or [--]), starting from the current node, the next element on the node list is read with [+], the previous with [-], the last with [++], and the first with [--]. Brackets cannot be followed by more brackets, e.g. name[++][-] is rejected with the return code KXL_RC_INVALID_NAME. If the specified name contains square brackets with contents other than '+','-', '++', '—', it will be interpreted as the index of an array.

If you are working with namespaces, the name parts with the corresponding prefix must be specified.

If the element to be read is an attribute node, the content of the attribute is returned through t_value.pValue. t_value.pType points to the empty character string.

If desired, the node is deleted after reading. Read with delete can only be executed on simple data elements at the lowest level, i.e. leaves of the XML object and attributes. If an entire subtree is to be deleted, there are two possibilities: Either all nodes must be deleted individually, or the root node of the subtree is overwritten with an arbitrary value, but of the same type specification (the subtree is thus implicitly deleted) and then read and deleted with Delete = KXL_TRUE.

The buffers pointed to by the pointers in the structure t_value are overwritten with the next KXLRead call. If necessary, the values must be saved in user-own storage areas.
If an error occurs, the returned t_value variable contains two NULL pointers.

Note: Attributes which define namespaces, cannot be read with these calls (see KXLSearchNS)

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_NODE_FOUND (1)	I	Node with the specified name not found or there are no further elements in the specified node list
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc; no reading possible
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_NO_SUBTREE_ DELETION (16)	U	In a read call with delete for a node, the node was not deleted, as it is not an leaf node. -> if the entire subtree is to be deleted, each node must be deleted individually.
KXL_RC_INVALID_NAME (19)	U	When the element name was analyzed, a syntax error in the use of "[", "]" "/" or @ was found. -> state a valid name
KXL_RC_CONVERSION_ERROR (22)	S/U	Code conversion error
KXL_RC_ATTR_NAME_ERROR (25)	U	Incorrect attribute name structure; no '/', '[' or ']' is allowed after '@'
KXL_RC_ATTR_READ_WRITE_ ERR (26)	U	Read or write error of an attribute
KXL_RC_NODE_TYPE_NOT_ ALLOWED (28)	U	For this call only an element node or attribute node is allowed.

where
I= Information, U=User error, S=System error

Example:

```
float Price;
Price = KXLToFloat(KXLRead(pNode, "price", KXL_FALSE, NULL), &RetCode);
```

If the element 'price' is not present, RetCode has the value KXL_RC_INVALID_T_VALUE.

4.6.2 Direct read

Function prototype:

```
t_value KXLReadNode(xmlNodePtr pNode ,int FullName, char** ppName,
                    short* pRetCode);
```

Parameters:

Type	Name	Comment	
xmlNodePtr	pNode	In:	Pointer to a node of the XML object.
int	FullName	In:	KXL_TRUE / KXL_FALSE
char**	ppName	Out:	Pointer to the name of the node.
short*	pRetCode	Out:	Return code, unless NULL was specified

Function result:

Type	Comment
t_value	t_value of the element

This function can be used to read an element of the XML object directly, without knowing the name. The name of the element and, in the structure t_value, the type and value of the element are returned from the current node of the XML object. If no type attribute is present, the empty string is returned as type. For interior nodes, the type corresponds to type="struct"; for leaf nodes, to type="string".
If FullName=KXL_TRUE is specified, the full pathname of the element (starting from the root node, but excluding this) is output; with FullName=KXL_FALSE, the name part corresponding to the read node is output. The name of the root node can therefore only be read with FullName=KXL_FALSE.
The name parts contain the namespace prefix, if there is one.

If an error occurs, NULL pointers are returned.
The buffers in which the data is returned (t_value, ppName) are overwritten with the next read call.

Notes:

1. The function will not recognize if the node that was read represents an element of a node list; it will also not recognize the position within a nodelist.
2. If an attribute is read, the returned attribute name starts with an '@'.

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object Possible cause: object destroyed or created incorrectly -> generate error documentation
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc; no reading possible
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_CONVERSION_ ERROR (22)	S/U	Code conversion error
KXL_RC_NODE_TYPE_NOT_ALL OWED (28)	U	For this call only an element node or attribute node is allowed.

where
I= Information, U=User error, S=System error

Example:

```
xmlNodePtr    pRoot;
t_value       tval;
char          Name[64];
char *        pName = &Name;
short         Retcode = KXL_RC_OK;
tval = KXLReadNode (pRoot, KXL_FALSE, &pName, &Retcode);
/* output of type, name and value of root node: */
sprintf("root node %s %s = %s \n", tval.pType, pName, tval.pValue);
```

4.6.3 Sequential read

Function prototype:

```
t_value KXLReadNextSib(xmlNodePtr pNode ,char** ppName,
                   xmlNodePtr* ppNode, short* pRetCode);
t_value KXLReadChild(xmlNodePtr pNode,char** ppName,
                   xmlNodePtr* ppNode, short* pRetCode);
t_value KXLReadNextSingleNode(xmlNodePtr pNode ,char** ppName,
                   xmlNodePtr* ppNode, short* pRetCode);
t_value KXLReadAttr(xmlNodePtr pNode ,char** ppName,
                   xmlNodePtr* ppNode, short* pRetCode)
```

Parameters:

Type	Name	Comment
xmlNodePtr	pNode	In: Pointer to a node of the XML object.
char**	ppName	Out: Pointer to the name of the node.
xmlNodePtr*	ppNode	Out: Pointer to the newly found node of the XML object.
short*	pRetCode	Out: Return code, unless NULL was specified

Function result:

Type	Comment
t_value	t_value of the element

These functions can be used to read all elements of the XML object sequentially, without the caller knowing the name. Starting from the current node, the name of the element, in the structure t_value, the type and value of the element, and the pointer to the read node are returned from the next node in the XML object. If no type attribute is present or an attribute is read, the empty string is returned as type. For interior nodes, type corresponds to type="struct"; for leaf nodes, to type="string".
If an error occurs, NULL pointers are returned. The nodes are read in the same sequence as they were written. Elements of an array, which are arranged in ascending order during writing, are an exception.
Depending on whether only the simple data elements (leaves) or attributes are to be read or all nodes (including structure and array nodes), different function calls are necessary.

KXLReadNextSib	reads the content of the next 'sibling' node (same level), If the current node is an attribute, further attributes are read sequentially.
KXLReadChild	reads the content of the (first) child node (one level deeper, no attributes)
KXLReadNextSingleNode	reads the content of the next leaf (lowest level, no attributes)
KXLReadAttr	reads the contents of the (first) attribute of the given node.

If KXLReadNextSingleNode is specified, the full pathname of the element (starting from the root node) is returned; otherwise, only the name part corresponding to the read node.

The name parts contain the namespace prefix, if there is one.
The buffers in which the data is returned (t_value, ppName) are overwritten with the next read call.

Notes:
1. The function will not recognize if the node that was read represents an element of a node list; it will also not recognize the position within a nodelist.
2. If an attribute is read, the returned attribute name starts with an '@'..
3. Attributes which define namespaces, cannot be read with these calls (see KXLSearchNS).

Return codes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_NODE_FOUND (1)	I	No further node found, i.e.: - with KXLReadNextSib: no sibling node in the current subtree or there are no further attributes - with KXLReadChild: no child node - with KXLReadNextSingleNode: no leaf (in the entire object) - with KXLReadAttr: no attributes present
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object Possible cause: object destroyed or created incorrectly -> generate error documentation
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc; no reading possible
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_CONVERSION_ERROR (22)	S/U	Error in code conversion
KXL_RC_NODE_TYPE_NOT_ALLOWED (28)	U	Only an element node can be specified with this call

where
I= Information, U=User error, S=System error

Examples:

1. Output all simple elements (leaves) of an object:

```
xmlNodePtr pRoot, pAct, pNew;
t_value    tval;
char Name[64];
char *     pName = &Name;
short Retcode = KXL_RC_OK;
pAct = pRoot;
while (1)
{ tval = KXLReadNextSingleNode (pAct, &pName, &pNew, &Retcode);
   if (Retcode == KXL_RC_NO_NODE_FOUND)
     break;
   /* output of Type, Name and Value of node: */
   sprintf("%s %s = %s \n", tval.pType, pName, tval.pValue);
   pAct = pNew;
}
```

2. Output all nodes of an object:

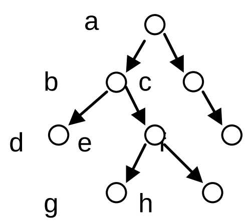
```
void ReadSubtree(xmlNodePtr pNode)
{   t_valuetval;
    char * pName;
    xmlNodePtr pAct;
    xmlNodePtr pNew;
    /* read child node */
    tval = KXLReadChild(pNode, &pName, &pNew, NULL);
    while (pNew != NULL)            /* no more children in subtree? */
    { /* output of Type, Name and Value of node */
sprintf("%s %s = %s \n", tval.pType, pName, tval.pValue);
      ReadSubtree (pNew);  /* recursive call: read subtree of child*/
      pAct = pNew;
      /* read next child */
      tval = KXLReadNextSib(pAct, &pName, &pNew, NULL);
    }
    return;
} /* end ReadSubtree */

xmlNodePtr  pRoot; /* root - Pointer of Object */
ReadSubtree(pRoot);
```

3. Output all attributes of an element:

```
xmlNodePtr pAct, pNew;
t_value    tval;
char       Name[64];
char       *     pName = &Name[0];
short            Retcode = KXL_RC_OK;
/* read first attribute */
tval = KXLReadAttr (pAct, &pName, &pNew, &Retcode);
while (Retcode == KXL_RC_OK)
{  /* output of name and value of node: */
   sprintf("%s = %s \n", pName, tval.pValue);
   pAct = pNew;
   /* read next attribute */
   tval = KXLReadNextSib (pAct, &pName, &pNew, &Retcode);
}
```

In the following object tree:



nodes d, g, h, f are read with example 1,
and nodes a, b, d, e, g, h, c, f with example 2

4.6.4 Read using local name

Function prototype:

```
t_value       KXLFindNode(xmlNodePtr pNode, char* pName, char** ppUrl,
              char** ppPrefix, xmlNodePtr* ppNode, short* pRetCode)
```

Parameters:

Type	Name	Comment
xmlNodePtr	pNode	In: Pointer to the node of the XML object.
char*	pName	In: Name of the element
char**	ppUrl	Out: Pointer to the Url of the newly found node of the XML object
char**	ppPrefix	Out: Pointer to the prefix of the newly found node of the XML object
xmlNodePtr*	ppNode	Out: Pointer to the newly found node of the XML object
short*	pRetCode	Out: Return code, unless NULL was specified

Function result:

Type	Comment
t_value	t_value of the elements

The function KXLFindNode is used to address an element specified by its local name, that means without regard to the namespace.
How to build an element name is described in section "Structure of element names, attribute names and node lists". A single element is specified by a node and the name relative to this node. The first element of a node list is read in the same way. If a node list position is specified ([+],[-],[++] or [--]), starting from the current node, the next element on the node list is read with [+], the previous with [-], the last with [++], and the first with [--]. Brackets cannot be followed by more brackets, e.g. name[++][-] is rejected with the return code KXL_RC_INVALID_NAME. If the specified name contains square brackets with contents other than '+','-', '++', '—', it will be interpreted as the index of an array.

If you are working with namespaces, all name parts except the last must be specified with the corresponding prefix. In contrast to KXLRead the function KXLFindNode doesn't allow to specify the last name part with namespace prefix.

KXLFindNode returns the type and value of the data item in the structure of type `t_value`, which can be converted using the functions described in section "Type conversion functions". If no type attribute is present, the empty string is returned as type. For inner nodes, the type corresponds to type="struct"; for leaf nodes, type="string". If the element to be read is an attribute node, the content of the attribute is returned through t_value.pValue. t_value.pType points to the empty string. If an error occurs, the returned t_value variable contains two NULL pointers.

Additional return values are ppUrl and ppPrefix which contain the pointer of the returned strings with the URL and the namespace prefix belonging to the newly read element. As the prefix of the default namespace the empty string is returned. If no namespace is assigned to the element and in case of error the empty string is returned for prefix and URL.
In ppNode the pointer to the newly read node is returned. In case of error the NULL pointer is returned.

As with writing, reading can be performed in two ways: If the root node is specified, a data item can be addressed via the fully qualified name. But reading can also be performed at multiple levels: KXLSetSubObject can be used to position in the XML object. Using the pointer to a subobject obtained in this way, KXLFindNode and the component name or array index (in the form [i]) can be used to access the individual data.
If an empty name is specified (spaces), the type and value of the specified node (pNode) are returned.

The buffers pointed to by the pointers ppUrl, ppPrefix and in the structure t_value are overwritten with the next KXLFindNode call. If necessary, the values must be saved in user-own storage areas.

Note: Attributes which define namespaces, cannot be read with KXLFindNode (see KXLSearchNS)

Returncodes:

Name (value)	Error type	Comment -> action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_NODE_FOUND (1)	I	Node with the specified name not found or there are no further elements in the specified node list
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc; no reading possible
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_INVALID_NAME (19)	U	When the element name was analyzed, a syntax error in the use of "[", "]" "/" or @ was found. -> state a valid name
KXL_RC_CONVERSION_ERROR (22)	S/U	Code conversion error
KXL_RC_ATTR_NAME_ERROR (25)	U	Incorrect attribute name structure; no '/', '[' or ']' is allowed after '@'
KXL_RC_ATTR_READ_WRITE_ ERR (26)	U	Read or write error of an attribute
KXL_RC_NODE_TYPE_NOT_ ALLOWED (28)	U	For this call only an element node or attribute node is allowed

where
I= Information, U=User error, S=System error

Example:

```
{   t_value tval;
    char ** ppUrl;
    char ** ppPrefix;
    xmlNodePtr pAct = pRoot;  // initiated with root node of document
    xmlNodePtr pNew;
    /* find all param child node */
    tval = KXLFindNode(pAct, "param", ppUrl, ppPrefix, &pNew, NULL);
    while (pNew != NULL)                   /* more children in subtree */
    {   /* output of name, value and namespace info of node */
        sprintf("value of {%s:%s}param = %s \n",
                ppPrefix, ppUrl, tval.pValue);
        /* read next param child node */
        tval = KXLFindNode(pNew, "[+]", ppUrl, ppPrefix, &pNew, NULL);
    }
    /* output of name, value and namespace info of node */
    sprintf("value of {%s:%s}param = %s \n", ppPrefix, ppUrl, tval.pValue);
```

For the document

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<application xmlns:utm="http://www.xml.test/test/utm"
         xmlns:fhs="http://www.xml.test/test/fhs"
         xmlns:uds="http://www.xml.test/test/uds"
         xmlns="http://www.xml.test/test/utm">
  <comment>openUTM application start parameter</comment>
  <param>START STARTNAME=TEST</param>
  <param>START FILEBASE=TEST.UTM</param>
  <utm:comment>format and database parameter</utm:comment>
  <fhs:param>MAPLIB=MAPLIB.FHS.TEST.UTM</fhs:param>
  <uds:param>DATABASE=UDSCONF</uds:param>
  <utm:comment>continue utm parameter</utm:comment>
  <utm:param>START ASYNTASKS=4</utm:param>
  <utm:param>START TASKS=5</utm:param>
</application>
```

the following output is given:

```
value of {:http://www.xml.test/test/utm}param = START STARTNAME=TEST
value of {:http://www.xml.test/test/utm}param = START FILEBASE=TEST.UTM
value of {fhs:http://www.xml.test/test/fhs}param = MAPLIB=MAPLIB.FHS.TEST.UTM
value of {uds:http://www.xml.test/test/uds}param = DATABASE=UDSCONF
value of {utm:http://www.xml.test/test/utm}param = START ASYNTASKS=4
value of {utm:http://www.xml.test/test/utm}param = START TASKS=5
```

4.6.5 Node list size query

Function prototype:

```
int     KXLGetSizeofNodelist(xmlNodePtr pNode, char *pName,
        short* pRetCode);
```

Parameters:

Type	Name	Comment	
xmlNodePtr	pNode	In:	Pointer to a node of the XML object.
char *	pName	In:	Name of the node list
short*	pRetCode	Out:	Return code, if NULL was specified

Function result:

Type	Comment
int	Number of elements of the node list

This function returns the number of elements of the node list with the name specified, which are sibling nodes to the specified node. The name must not contain any structural elements, such as /, [,], @. The namespace prefix must be specified, where present (see the section on namespaces).
If the NULL pointer or an empty character string is specified as the name, the size of the node list with the name of the current node is found.

Return codes:

Name (value)	Error type	Comment -> Action
KXL_RC_OK (0)	I	Function carried out correctly
KXL_RC_NO_NODE_FOUND (1)	I	No element of the specified node list was found
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_INVALID_NAME (19)	U	When the element name was analyzed, a syntax error in the use of "[", "]" "/" or @ was found -> state a valid name

where
I= Information, U=User error, S=System error

4.7 Handling character sets

4.7.1 Reading the home character set

Function prototype:
```
char*   KXLGetHomeEnc(short* pRetCode);
```

Parameters:
```
Type    Name         Comment
short*  pRetCode     Out:  Return code, unless NULL was specified
```

Function result:
```
Type    Comment
char*   (printable) home encoding
```

This function returns the name of the current home encoding.

Return codes:

Name (value)	Error type	Comment -> Action
KXL_RC_OK (0)	I	function has been executed successfully

where
I= Information, U=User error, S=System error

4.7.2 Reading the document character set

Function prototype:
```
char*   KXLGetDocEnc(xmlNodePtr pNode, short* pRetCode);
```

Parameters:
```
Type       Name           Comment
xmlNodePtr pNode    In:    Pointer to a node of the XML object
short*    pRetCode  Out:   Return code, unless NULL was specified
```

Function result:
```
Type    Comment
char*   Name of the document encoding
```

This function returns the encoding name of the document to which the specified node belongs. At the next call, the return value is overwritten.

Return codes:

Name (value)	Error type	Comment -> Action
KXL_RC_OK (0)	I	function carried out successfully
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object. Possible cause: object destroyed or created with errors -> generate error documentation
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer

where
I= Information, U=User error, S=System error

4.7.3 Changing the document character set

Function prototype:
```
void    KXLSetDocEnc(xmlNodePtr pNode, char* pEnc, short* pRetCode);
```

Parameters:
```
Type       Name       Comment
xmlNodePtr pNode      In:   Pointer to a node of the XML object
char*      pEnc       In:   Name of the document encoding
short*     pRetCode   Out:  Return code, unless NULL was specified
```

The function changes the encoding statement of the object, to which the specified node belongs. It also checks if this encoding statement is known to the parser. The encoding statement of the document is deleted if `pEnc` is NULL or if it points to the empty character string.

Return codes:

Name (value)	Error type	Comment-> Action
KXL_RC_OK (0)	I	function carried out successfully
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object. Possible cause: object destroyed or created with errors -> generate error documentation
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer
KXL_RC_UNKNOWN_ENCODING (30)	U	The specified encoding is not known to the parser -> declare the conversion routines of the encoding to the parser

where
I= Information, U=User error, S=System error

4.7.4 Converting a character string to UTF-8

Function prototype:
```
char*   KXLStringToUTF8(char* pString, short* pRetCode);
```

Parameters:
```
Type       Name       Comment
char*      pString    In:   Character string in the home encoding
short*     pRetCode   Out:  Return code, unless NULL was specified
```

Function result:
```
Type    Comment
char*   Character string in UTF-8
```

The function converts the specified character string from home encoding to UTF-8. It also checks if the encoding statement is known to the parser.
When the first call is issued, the KXLInitEnv function may be called implicitly (see section 4.1 Initialization of the UTM XML interface).

Note: The buffer where the converted character string is returned, is overwritten with the next KXLStringToUTF8 call.

Return codes:

Name (value)	Error type	Comment-> measure
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_VERSION_ERR (23)	U	The versions of the UTM XML components are inconsistent
KXL_RC_PARSER_VERS_NOT_ SUPP (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate newer parser version
KXL_RC_UNKNOWN_ENCODING (30)	U	The specified encoding is not known to the parser -> declare the conversion routines of the encoding to the parser
KXL_RC_INVALID_PARAMETER (32)	U	Incorrect parameter statement, e.g. NULL pointer specified, although not allowed
KXL_RC_ENCODING_CHANGE_ERR (36)	U/S	No encoding handler could be set up for the home encoding

where
I= Information, U=User error, S=System error

4.7.5 Converting a character string from UTF-8 to the home encoding

Function prototype:
```
char*  KXLStringFromUTF8(char* pString, short* pRetCode);
```

Parameters:
```
Type     Name      Comment
char*    pString   In:   Character string in UTF-8
short*   pRetCode  Out:  Return code, unless NULL was specified
```

Function result:
```
Type   Comment
char*  Character string in home encoding
```

The function converts the specified character string from UTF-8 to the home encoding. It also checks if the encoding statement is known to the parser.
When the first call is issued, the KXLInitEnv function may be called implicitly (see section 4.1 Initialization of the UTM XML interface).
Note: The buffer where the converted character string is returned, is overwritten with the next KXLStringFromUTF8 call.

Return codes:

Name (value)	Error Type	Comment-> measure
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_VERSION_ERR (23)	U	The versions of the UTM XML components are inconsistent
KXL_RC_PARSER_VERS_NOT_ SUPP (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate newer parser version
KXL_RC_UNKNOWN_ENCODING (30)	U	The specified encoding is not known to the parser -> declare the conversion routines of the encoding to the parser
KXL_RC_INVALID_PARAMETER (32)	U	Incorrect parameter statement , e.g. NULL pointer specified, although not allowed

where
I= Information, U=User error, S=System error

4.7.6 Reading the original encoding of a given alias name

Function prototype:
```
char*  KXLGetEncodingAlias(char* alias, short* pRetCode);
```

Parameters:
```
Type     Name      Comment
char*    alias     In:   Alias name of an encoding
short*   pRetCode  Out:  Return code, unless NULL was specified
```

Function result:
```
Type   Comment
char*  Original encoding name
```

This function searches for the encoding that is assigned to the given alias name. The name of the original encoding will be returned or NULL if the alias name is not known.

In some cases the function KXLInitEnv might be called internally on the first call of this function (see section 4.1 Initialization of the UTM XML interface).

Notes:
1. The buffer in which the converted string is returned will be overwritten on the next call of KXLGetEncodingAlias.
2. For the handling of different character sets and their names see section 3.9 Character Sets/Encoding.

Return codes:

Name (value)	Error type	Comment -> Measure
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_ALLOC_ERROR (11)	S	No additional space could be requested with malloc/realloc
KXL_RC_CONVERSION_ERROR (22)	S/U	Error in code conversion
KXL_RC_VERSION_ERR (23)	U	The versions of the UTM XML components are inconsistent
KXL_RC_PARSER_VERS_NOT_ SUPP (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate new parser version
KXL_RC_UNKNOWN_ENCODING (30)	U	The specified encoding is not known to the parser -> declare the conversion routines of the encoding to the parser
KXL_RC_INVALID_PARAMETER (32)	U	Incorrect parameter statement , e.g. NULL pointer specified although not allowed

where
I= Information, U=User error, S=System error

4.7.7 Assigning an alias name to an original encoding

Function prototype:
```
void   KXLSetEncodingAlias(char* alias, char* encName, short* pRetCode);
```

Parameter:
```
Type      Name       Comment
char*     alias      In:   Alias name of an encoding
char*     encName    In:   Original name of an encoding
short*    pRetCode   Out:  Return code, unless NULL was specified
```

This function assigns the given alias name to the specified original encoding. If `encName` is NULL the current assignment of the given alias name is deleted.

In some cases the function KXLInitEnv is called internally on the first call of this function (see section 4.1 Initialization of the UTM XML interface).

Notes:
1. For `encName` only a defined original encoding name can be specified. Alias names are rejected.
2. If for `alias` the name of a defined original encoding is specified, documents with this encoding are processed with the encoding handler of the encoding assigned to `alias`.
3. The declaration of alias names is processed task/process specific. When using the UTM-XML interface in an UTM application, this means that this call must be given in each task/process, e.g. in the start exit.
4. For handling different character sets and their names see section 3.9 Character Sets/Encoding.

Return codes:

Name (value)	Error type	Comment -> Measure
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_ALLOC_ERROR (11)	S	No additional space could be requested with malloc/realloc
KXL_RC_CONVERSION_ERROR (22)	S/U	Error in code conversion
KXL_RC_VERSION_ERR (23)	U	The versions of the UTM XML components are inconsistent
KXL_RC_PARSER_VERS_NOT_ SUPP (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate new parser version
KXL_RC_UNKNOWN_ENCODING (30)	U	The specified encoding is not known to the parser -> declare the conversion routines of the encoding to the parser
KXL_RC_INVALID_PARAMETER (32)	U	Incorrect parameter statement , e.g. NULL pointer specified although not allowed
KXL_RC_PARSER_ENCODING_E RR (46)	S/U	Parser couldn't execute the function -> check, if alias name and, if needed, encoding name are defined correctly.
KXL_RC_IS_ALIAS_ENCODING (47)	U	Given encoding name is alias name -> specify original encoding name

where
I= Information, U=User error, S=System error

4.8 Namespace management

4.8.1 Writing a namespace definition

Function prototype:

```
xmlNsPtr      KXLWriteNS (xmlNodePtr pNode, char * prefix, char * url,
                           short * pRetCode )
```

Parameters:

```
Type       Name       Comment
xmlNodePtr pNode      In:   pointer to a node of the XML object.
char*      prefix     In:   prefix of the namespace definition
char*      url        In:   URL of the namespace definition
short*     pRetCode   Out:  Return code, unless NULL was specified
```

Function result:

```
Type    Comment
xmlNsPtr      Pointer to the written namespace definition
```

With the function KXLWriteNs, a namespace can be defined or redefined. i.e. a particular URL `url` is assigned to a namespace prefix `prefix`. This namespace definition is assigned to a specific element node `pNode` of the XML object.

`Prefix` and `url must` not be NULL pointer. To define a default namespace, enter an empty character string for `prefix`.

If a namespace definition with the specified prefix has already been assigned to this node, this is overwritten. If no namespace definition with the specified prefix has been assigned to this node, a new namespace entry is set up, and the whole subtree is searched for elements and attributes with this prefix. If they have been assigned to a dummy definition, these are replaced with references to the new namespace definition.

The entry to which the returned pointer points, contains references to UTF-8 character strings. For converting them to home encoding the function KXLStringFromUTF8 is available.

For more information on using namespaces and the areas in which they are valid, see the section on namespaces.

Return codes:

Name (value)	Error Type	Comment-> measure
KXL_RC_OK (0)	I	function carried out successfully
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc, cannot read
KXL_RC_NO_XML_NODE (15)	U	the pointer to the current node is NULL -> specify valid pointer
KXL_RC_CONVERSION_ ERROR (22)	S/U	code conversion error
KXL_RC_NODE_TYPE_NOT_ ALLOWED (28)	U	only an element node can be specified with this call
KXL_RC_INVALID_PARAMETER (32)	U	incorrect parameter statement , e.g. NULL pointer specified, although not allowed
KXL_RC_NAMESPACE_WRITE_ERR OR (33)	S/U	the parser could not write the namespace definition correctly, see parser error messages

where
I= Information, U=User error, S=System error

4.8.2 Erasing a namespace definition

Function prototype:
```
xmlNodePtr KXLDelNS (xmlNodePtr pNode, char * prefix, short * pRetCode )
```

Parameters:
```
Type       Name       Comment
xmlNodePtr pNode      In:   Pointer to a node of the XML object.
char*      prefix     In:   Prefix of the namespace definition
short*     pRetCode   Out:  Return code, unless NULL was specified
```

Function result:
```
Type          Comment
xmlNodePtr    Pointer to a node of the XML object or NULL
```

A namespace with the prefix `prefix`, which is assigned to a specific element node, can be deleted with the function KXLDelNs.

`Prefix` must not be the NULL pointer. To delete a default namespace, enter an empty character string as prefix. If there is no namespace definition with the specified prefix in this node, a corresponding return code is returned.

If there is an element node or attribute node in the subtree of the specified node, which refers to the namespace, which is to be deleted, the deletion is not carried out, and the pointer of the node which has been found is returned. Otherwise the NULL pointer is returned.

For more information on the use of namespaces, see the section on namespaces.

Return codes:

Name (value)	Error Type	Comment-> measure
KXL_RC_OK (0)	I	function carried out successfully
KXL_RC_ALLOC_ERROR (11)	S	no more new storage space could be requested with malloc/realloc, cannot read
KXL_RC_NO_XML_NODE (15)	U	the pointer to the current node is NULL -> specify valid pointer
KXL_RC_CONVERSION_ ERROR (22)	S/U	code conversion error
KXL_RC_NODE_TYPE_NOT_ ALLOWED (28)	U	only an element node is allowed with this call
KXL_RC_NAMESPACE_NOT_ FOUND (31)	I	namespace not found
KXL_RC_INVALID_PARAMETER (32)	U	incorrect parameter statement , e.g. NULL pointer specified, although not allowed
KXL_RC_NAMESPACE_DEL_ ERROR (34)	U	the definition cannot be deleted, as at least one element node or attribute node still refers to it.

where
I= Information, U=User error, S=System error

4.8.3 Reading a list of namespace definitions

Function prototype:

```
xmlNsPtr*     KXLReadNSList (xmlNodePtr pNode, short * pRetCode )
```

Parameters:

Type	Name	Comment
xmlNodePtr	pNode	In: Pointer to a node of the XML object.
short*	pRetCode	Out: Return code, unless NULL has been specified.

Function result:

Type	Comment
XmlNsPtr*	Pointer to an array of namespace definitions.

The function KXLReadNsList can produce an output listing all the relevant namespace definitions for a specific element node `pNode`. I.e. it returns an array of pointers to the namespace definitions, which are valid for the specified node – these include not just namespace definitions set within the node itself, but also definitions from parent nodes. Dummy entries (with empty url/href) are also returned - the structure of these entries is defined in the parser include file libxml/tree.h as structure xmlNs.

The entries indicated by the returned pointer contain references to UTF-8 character strings. The function KXLStringFromUTF8 can be used to convert them into home encoding.
For more information on the use of namespaces, see the section on namespaces.

Caution: The array of pointers to namespace definitions, which is returned by the call, must be explicitly released by the user with xmlFree (parser function).

Return codes:

Name (value)	Error Type	Comment-> measure
KXL_RC_OK (0)	I	function carried out successfully
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object; Possible cause: object destroyed or created incorrectly -> generate error documentation
KXL_RC_NO_XML_NODE (15)	U	the pointer to the current node is NULL -> specify valid pointer
KXL_RC_NODE_TYPE_NOT_ ALLOWED (28)	U	only an element node can be specified with this call
KXL_RC_NAMESPACE_NOT_ FOUND (31)	I	namespace not found

where
I= Information, U=User error, S=System error

Example:

```
xmlNodePtr pNode;
xmlNsPtr * pNScur;
int i;
short rcode;
pnode = …;
/* read all namespaces valid for pNode */
pNScur = KXLReadNSList (pNode, &rcode);
if (rcode == KXL_RC_OK)
{ /* build output */
  printf ("namespaces found:\n");
  for (i=0;pNScur[i] != NULL; i++)
  { /* loop over all namespaces found */
    if (pNScur[i]->prefix == NULL)
      printf ("<default> : ");
    else
      printf ("%s  : ", pNScur[i]->prefix);
    printf ("%s \n", pNScur[i]->href);
  }
```

4.8.4 Finding a namespace definition

Function prototype:

```
xmlNsPtr      KXLSearchNS (xmlNodePtr pNode, char * prefix, char * url,
                           short * pRetCode )
```

Parameters:

```
Type       Name       Comment
xmlNodePtr pNode      In:   Pointer to a node of the XML object.
char*      prefix     In:   Prefix of the namespace definition
char*      url        In:   URL of the namespace definition
short*     pRetCode   Out:  Return code, unless NULL was specified
```

Function result:

```
Type    Comment
xmlNsPtr      Pointer to the namespace definition which has been found
```

The function KXLSearchNs searches for a relevant namespace definition for a specific element node `pNode.`

If the URL `url` is specified, the function searches for the namespace definition with the URL specified. If the NULL pointer is specified as `url`, it looks for the namespace definition with the specified prefix `prefix`. If this is also empty, or the NULL pointer, it looks for the default namespace which is valid for the element node. The return value is the pointer to the namespace entry which is found or NULL if no suitable namespace is found. Namespace entries with empty URL are not returned.

The structure of one of these namespace entries is defined in the parser include file libxml/tree.h as structure xmlNs.

The entry, which the returned pointer indicates, contains references to UTF-8 character strings. The function KXLStringFromUTF8 can be used to convert them into home encoding.

For more information on the use of namespaces, see the section on namespaces.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_INVALID_DOC_PTR (10)	S/U	The stated object node is not assigned to any object. Possible cause: object destroyed or created with errors -> generate error documentation
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> state valid pointer
KXL_RC_CONVERSION_ ERROR (22)	S/U	Error in code conversion
KXL_RC_NODE_TYPE_NOT_ ALLOWED (28)	U	Only an element node can be stated with this call
KXL_RC_NAMESPACE_NOT_ FOUND (31)	I	namespace not found
KXL_RC_INVALID_PARAMETER (32)	U	Incorrect parameter specified , e.g. NULL pointer specified, although not allowed

where
I= Information, U=User error, S=System error

4.9 XML schema validation

4.9.1 Converting an XML document in an XML object with schema validation

Function prototype:

```
xmlNodePtr    KXLConvDocToObjAndValid (char* pBuffer, int validate, char*
pDir,                                     short* pRetCode);
```

Parameters:

```
Type    Name         Comment
char*   pBuffer      In:   Buffer with the XML document
int     validFlag    In:   Specifies if validation is to be performed
char*   pDir         In:   Name of the directory for file referencing
short*  pRetCode     Out:  Return code when NULL was not specified
```

Function result:

```
Type          Comment
xmlNodePtr    Pointer to the root node of the XML object
```

For processing an XML document it must be converted to an XML object. The KXLConvDocToObjAndValid function can be called for this purpose. The buffer with the entire (\0-terminated!) XML document is passed as an input parameter of the call. If desired, the XML document can be validated against the first schema specified in the document in schemaLocation. The function returns a pointer to the root node of the XML object. NULL is returned in the case of error, e.g. when a document is invalid.

If a document to be converted contains an encoding specification, then this is stored in the object tree and all names and contents are converted to UTF-8 and stored in the object tree. If there is no encoding specification, then UTF-8 is assumed. A code conversion routine (to and from UTF-8) must be declared for the encoding specification entered, otherwise an error is returned and no object is created (see the Character set section for more information).

If the document contains a schemaLocation element and the validate parameter is not 0, then the XML document is validated against the first XML schema specified in schemaLocation. During validation, all URLs specified (meaning those specified in schemaLocation and in include, import, redefine…) are converted to local file names when the pDir parameter is not NULL (see section Entities Loader). The schema is parsed internally, but the memory used is freed when the function terminates. This means that the schema cannot be processed further.
If the XML document is not well-formed or not a valid instance of the schema, then the corresponding return code is returned (return codes 20, 38, 39, 44). You can get more detailed information on the error by calling KXLGetLastParserError or by looking in the trace file.

if the validate parameter = 0, then the functionality is the same as in the KXLConvDocToObj function.

When the first call is issued, the KXLInitEnv function may be called implicitly (see section 4.1 Initialization of the UTM XML interface). The corresponding return code is returned and no object is created in case of an error.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function executed correctly; the document is an instance of the specified schema
KXL_RC_NO_NODE_FOUND (1)	I	SchemaLocation specification not found
KXL_RC_NO_MALLOC_FOR_ PARSER_1(5)	S	Not enough memory space available for the object -> increase program memory
KXL_RC_NO_MALLOC_FOR_	S	Not enough memory space available for the object

Name (value)	Error Type	Comment-> Action
PARSER_2(6)		-> increase program memory
KXL_RC_NO_CODE_ CONVERSION (8)	U	The document does not start with the XML start tag '<?xml' or '<?XML' in EBCDIC or ASCII -> correct the document and/or convert the code
KXL_RC_EMPTY_DOC (9)	U	The converted object does not contain any nodes
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_NO_XML_NODE (15)	S	internal error while reading the SchemaLocation
KXL_RC_INVALID_NAME (19)	S	internal error while reading the SchemaLocation
KXL_RC_PARSER_ERROR (20)	S/U	The parser has detected an error -> see the message from the parser. (trace file or KXLGetLastParserError)
KXL_RC_NO_CONTEXT_FOR_ PARSER (21)	S	Error while creating internal parser management spaces -> eliminate any eventual memory bottlenecks
KXL_RC_CONVERSION_ERROR (22)	S/U	Error during code conversion
KXL_RC_VERSION_ERR (23)	U	The versions of the UTM XML components are inconsistent
KXL_RC_PARSER_VERS_NOT_ SUPP (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate newer parser version
KXL_RC_NODE_TYPE_NOT_)WED (28)	S	internal error while reading the SchemaLocation
KXL_RC_UNKNOWN_ ENCODING (30)	U	Encoding name is unknown to the parser -> declare the conversion routines for encoding in the parser
KXL_RC_NAMESPACE_NOT_ FOUND (31)	I	Namespace searched for not found
KXL_RC_INVALID_PARAMETER (32)	U	Parameter specification incorrect, e.g. a NULL pointer was specified but is not allowed
KXL_RC_ENCODING_CHANGE_ ERR (36)	U/S	No encoding handler could be set up for the specified home encoding
KXL_RC_PARSER_CONTEXT_ ERR (37)	S	Internal parser error while creating internal parser management spaces -> eliminate any eventual memory bottlenecks
KXL_RC_SCHEMA_PARSE_ INT_ERR (38)	U	Error while parsing the schema -> see the message or return code from the parser. (trace file or KXLGetLastParserError)
KXL_RC_SCHEMA_VALID_INT_ ERR (39)	U	Error while validating the schema -> see the message or return code from the parser. (trace file or KXLGetLastParserError)
KXL_RC_NO_SCHEMA_GIVEN (40)	U	The schema input address is NULL
KXL_RC_DOC_NOT_VALID (44)	U	The document is not a valid instance of the specified schema -> see the message from the parser. (trace file or KXLGetLastParserError)

where
I= Information, U=User error, S=System error

4.9.2 Parsing an XML schema stored in memory

Function prototype:
```
xmlSchemaPtr  KXLParseSchema(char* pBuffer, char* pDir, short* pRetCode);
```

Parameters:
```
Type    Name         Comment
char*   pBuffer      In:   Buffer with the XML schema document
char*   pDir         In:   File directory for referenced files
short*  pRetCode     Out:  Return code when NULL was not specified
```

Function result:
```
Type          Comment
xmlSchemaPtr  Pointer to the XML schema structure
```

This function checks if the input buffer contains a valid XML schema. During validation, a document tree is generated internally for this schema, and other administration data is stored in an internal XML schema structure. If the pDir parameter is not NULL (see 3.11.1 Entities Loader), then all URLs specified, e.g. in an include or import statement, are converted to local file names. With the address returned in xmlSchemaPtr the validation of an XML document against this schema can be initiated by the call of KXLValidDoc or KXLValidDocBuf.

If the XML schema is not well-formed, then the corresponding return code is returned (KXL_RC_SCHEMA_PARSE_INT_ERR (38)). You can request more detailed information on the error via KXLGetLastParserError or by looking in the trace file.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_EMPTY_DOC(9)	U	Empty input buffer
KXL_RC_VERSION_ERR (23)	U	The versions of the UTM XML components are inconsistent
KXL_RC_PARSER_VERS_NOT_ SUPP (24)	U	Old version of XML parser integrated (< 2.6.20) -> integrate newer parser version
KXL_RC_UNKNOWN_ ENCODING (30)	U	Encoding name is unknown to the parser -> declare the conversion routines for encoding in the parser
KXL_RC_INVALID_PARAMETER (32)	U	Address of the input buffer is NULL
KXL_RC_ENCODING_CHANGE_ ERR (36)	U /S	No encoding handler could be set up for the specified home encoding
KXL_RC_PARSER_CONTEXT_ ERR (37)	S	Internal parser error while creating internal parser management spaces -> eliminate any eventual memory bottlenecks
KXL_RC_SCHEMA_PARSE_ INT_ERR (38)	U	Error while parsing the schema -> see the message or return code from the parser. (trace file or KXLGetLastParserError)

where
I= Information, U=User error, S=System error

4.9.3 Parsing an XML schema from a file

Function prototype:

```
xmlSchemaPtr  KXLParseSchemaFile(char* pFilename, char* pDir, short*
pRetCode);
```

Parameters:

```
Type    Name         Comment
char*   pFilename    In:   File with the XML schema document
char*   pDir         In:   File directory for referenced files
short*  pRetCode     Out:  Return code when NULL was not specified
```

Function result:

```
Type          Comment
xmlSchemaPtr  Pointer to the XML schema structure
```

This function checks if the file specified in pFilename contains a valid XML schema. During validation, a document tree is generated internally for this schema, and other administration data is stored in the internal XML schema structure. If the pDir parameter is not NULL (see 3.11.1 Entities Loader), then all URLs specified, e.g. in an include or import statement, are converted to local file names. The file name pFilename is not redirected to pDir. With the address returned in xmlSchemaPtr the validation of an XML document against this schema can be initiated by the call of KXLValidDoc or KXLValidDocBuf.

If the XML schema is not well-formed, then the corresponding return code is returned (KXL_RC_SCHEMA_PARSE_INT_ERR (38)). You can get more detailed information on the error by calling KXLGetLastParserError or by looking in the trace file.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_VERSION_ERR (23)	A	The versions of the UTM XML components are inconsistent
KXL_RC_PARSER_VERS_NOT_ SUPP (24)	A	Old version of XML parser integrated (< 2.6.20) -> integrate newer parser version
KXL_RC_INVALID_PARAMETER (32)	A	Address of the file name is NULL
KXL_RC_PARSER_CONTEXT_ ERR (37)	S	Internal parser error while creating internal parser management spaces -> eliminate any eventual memory bottlenecks
KXL_RC_SCHEMA_PARSE_ INT_ERR (38)	A	Error while parsing the schema -> see the message or return code from the parser. (trace file or KXLGetLastParserError)

where
I= Information, U=User error, S=System error

4.9.4 Validating an XML object against an XML schema

Function prototype:

```
void   KXLValidDoc(xmlNodePtr pNode, xmlSchemaPtr pSchema, short* pRetCode);
```

Parameters:

Type	Name	Comment
xmlNodePtr	pNode	In: Pointer to a node of the XML object
xmlSchemaPtr	pSchema	In: Pointer to the XML schema structure
short*	pRetCode	Out: Return code (mandatory!)

Function result:

None

This function checks if the XML document represented by the XML object specified in xmlNodePtr is an instance of the XML schema defined in xmlSchemaPtr. A return code field must be specified for this function.

If an error occurs during validation, then the corresponding return code is returned (return codes 39, 44). You can request more detailed information on the error by calling KXLGetLastParserError or by looking in the trace file.

If the mandatory parameter pRetcode is not specified (=NULL), then an error message is output to the trace file.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to an object; possible reason: object destroyed or created incorrectly -> compile error documents
KXL_RC_NO_XML_NODE(15)	U	The pointer to the current node is NULL -> specify a valid pointer
KXL_RC_PARSER_CONTEXT_ ERR (37)	S	Internal parser error while creating internal parser management spaces -> eliminate any eventual memory bottlenecks
KXL_RC_SCHEMA_VALID_INT_ ERR (39)	U	Error during validation -> see the message or return code from the parser. (trace file or KXLGetLastParserError)
KXL_RC_NO_SCHEMA_GIVEN (40)	U	The schema input address is NULL
KXL_RC_DOC_NOT_VALID(44)	U	The document is not a valid instance of the specified schema -> see the message from the parser. (trace file or KXLGetLastParserError)

where
I= Information, U=User error, S=System error

4.9.5 Validating an XML document in memory against an XML schema

Function prototype:

```
void   KXLValidDocBuf(char* pDocBuf, xmlSchemaPtr pSchema, short* pRetCode);
```

Parameters:

```
Type          Name        Comment
char*         pDocBuf     In:   Buffer containing the XML document
xmlSchemaPtr  pSchema     In:   Pointer to the XML schema structure
short*        pRetCode    Out:  Return code (mandatory!)
```

Function result:

```
none
```

This function checks if the XML document in the buffer pDocBuf is an instance of the XML schema defined in xmlSchemaPtr. A return code field must be specified for this function.

If an error occurs during validation, then the corresponding return code is returned (return codes 39, 44). You can request more detailed information on the error by calling KXLGetLastParserError or by looking in the trace file.

If the mandatory parameter pRetcode is not specified (=NULL), then an error message is output to the trace file.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_EMPTY_DOC(9)	U	Empty input buffer
KXL_RC_INVALID_PARAMETER (32)	U	The address of the input buffer is NULL
KXL_RC_PARSER_CONTEXT_ ERR (37)	S	Internal parser error while creating internal parser management spaces -> eliminate any eventual memory bottlenecks
KXL_RC_SCHEMA_VALID_INT_ ERR (39)	U	Error during validation -> see the message or return code from the parser. (trace file or KXLGetLastParserError)
KXL_RC_NO_SCHEMA_GIVEN (40)	U	The schema input address is NULL
KXL_RC_DOC_NOT_VALID(44)	U	The document is not a valid instance of the specified schema -> see the message from the parser. (trace file or KXLGetLastParserError)

where
I= Information, U=User error, S=System error

4.9.6 Freeing a schema tree in memory

Function prototype:

```
void   KXLFreeSchema(xmlSchemaPtr pSchema, short* pRetCode);
```

Parameters:

Type	Name	Comment
`xmlSchemaPtr`	`pSchema`	`In:   Pointer to the XML schema structure`
`short*`	`pRetCode`	`Out:  Return code when NULL was not specified`

Function result:

`None`

This call frees the memory required for the XML schema. This function should always be called when you are done working with the corresponding XML schema.
After freeing the memory, you must not access the addresses of the freed schema structure or the XML objects of the schema.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function carried out successfully
KXL_RC_NO_SCHEMA_GIVEN (40)	u	The schema input address is NULL

where
I= Information, U=User error, S=System error

4.10 Diagnostic functions

4.10.1 Trace initialization

Function prototype:

```
    KXLTSENV();
```

Parameters:

none

Function result:

none

The function KXLTSENV initializes the trace function of the present XML interface. Depending on the trace mode setting (in the job variable or shell variable or environment variable KXLTRAC), trace records are written to files which can be analyzed in the event of an error.

The call is implicitly issued with each KXLCreateNewObj, KXLConvDocToObj and KXLParseSchema call, i.e. whenever a new object is created. However, it can also be issued explicitly. In this case the header file `libxml/kxltrace.h` must be specified.

The trace is initialized on a process- or task-specific basis, i.e. each process/task must issue a trace initialization call before the writing XML trace records.

Information about setting the trace mode, and the structure of file names and trace records, are given in chapter "Diagnostics", section "Trace".

Note for BS2000:
In BS2000, the job variable with the link name KXLTRAC is read each time KXLTSENV is called. If the trace mode has been changed in the meantime, the new mode is registered in the program.

4.10.2 Information on the last parser error

Function prototype:

```
int     KXLGetLastParserError(char** ppErrmsg, short* pRetCode);
```

Parameters:

Type	Name	Comment
char **	ppErrmsg	Out: Pointer to the last parser error message
short*	pRetCode	Out: Return code, unless null has been specified

Function result:

Type	Comment
int	Parser return code

If an error is recognized in the parser (during parsing, validating or another call), the parser writes an error code internally and/or outputs a message (to trace file), depending on the context in which the error was recognized. The caller receives one of the error return codes listed below caused by an parser detected error. With the call KXLGetLastParserError, the caller receives the last parser RC or the last parser message. The field which was not supplied is overwritten with 0 (return code) or '\0' (message). The parser return codes are defined in the include file libxml/xmlerror.h under xmlParserErrors.

List of return codes from parser errors:
- KXL_RC_NO_ROOT_CREATED (4)
- KXL_RC_NO_CONVERSION_TO_DOC (7)
- KXL_RC_NO_CHILD_CREATED (13)
- KXL_RC_NO_TYPE_ATTR_FOUND(14)
- KXL_RC_NO_XML_NODE (15)
- KXL_RC_PARSER_ERROR (20)
- KXL_RC_NO_CONTEXT_FOR_PARSER (21)
- KXL_RC_CONVERSION_ERROR (22)
- KXL_RC_ATTR_READ_WRITE_ERR (26)
- KXL_RC_PARSER_CONTEXT_ERR (37)
- KXL_RC_SCHEMA_PARSE_INT_ERR (38)
- KXL_RC_SCHEMA_VALID_INT_ERR (39)
- KXL_RC_DOC_NOT_VALID (44)

Comment:
If the UTM XML trace is initialized, messages are also displayed in the trace file. If several parser messages were displayed after one error, it is worth looking them up in the trace file.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function carried out successfully.

where:
I= Information, U=User error, S=System error

5 XML API for COBOL

5.1 General

As the C interface cannot be mapped in unchanged form onto Cobol, an adapter module called KXLCOB.c is available with its own interface, called KXLFUNC (described below), which is called by Cobol subroutines.
To call the interface, the Copy element `KXLCOBOL` must be specified in the program. It contains the definition of the parameter block and the condition names.

Apart from the following special features of the Cobol interface, the calls, their parameters and return values have the same meaning as under the C API and are described in detail there.

5.1.1 Parameters

A structure conforming to the definition of KC-XML-PARAMETER is passed as the first parameter area. This is available in the Copy element `KXLCOBOL` and must be specified in all calls of the interface. Further parameters (character strings) must be specified, depending on the OPCODE.

The following applies to all calls:

- As input parameter KC-XML-PARAMETER with VERSION= 1 must be supplied
- RTCODE in KC-XML-PARAMETER is always returned

This means that in the Cobol API, unlike with the C API, it is not possible to dispense with the return of the return code.

All length fields (required according to the function call) in the parameter area must be supplied with values when the Cobol interface is called. This is except for field BUFFER-RETURN-LTH, in which KXLConvObjToDoc returns the real length of the converted document to the Cobol program and field TYPE-RETURN-LTH in the KXLReadXxx read calls, in which the length of the returned type string is returned.
When lengths < 0 are specified, a corresponding return code is returned.

5.1.2 Passing character strings

As Cobol and C have different definitions of character strings (Cobol character strings with fixed lengths, possibly right-filled with blanks, C character strings with terminating '/0' (a byte with value X'00')), the adapter module behaves as follows:

- Character strings received from Cobol are copied, minus the "padded" blanks, into a new buffer and terminated by '/0'. The C character strings obtained in this way are passed on to the C interface.
- The character strings returned by the C interface are copied (without the terminating null byte) into the character string variable provided by Cobol. Depending on the length of the C character string and the length specified for this variable in the parameter area, the string is padded with blanks or truncated.
- If a NULL pointer or the empty character string is expected at the C interface, the length field must be preset to 0 by the Cobol program. If the NULL pointer is returned by the C interface in the event of an error, the return field is not modified by the Cobol program, and the return code must be evaluated. If empty values are returned ("emptystring" if, say, no type attribute or no content is present), the

corresponding return field is filled with blanks and the TYPE-RETURN-LTH field supplied with 0 if necessary.

5.1.3 Condition variables

Certain variables in the parameter area are defined as condition variables. They are addressed as follows with the associated condition names (88 data elements):

- Set, for example, OPCODE for KXLCreateOBJECT: SET KC-XML-CREATE-OBJECT TO TRUE.
- Queries: IF KC-XML-CREATE-OBJECT THEN ...

5.2 Return values

The following return values are returned in addition by the Cobol API:

Name (value)	Error type	Comment -> action
KXL_RC_COBOL_PARAM_ ERROR(50)	U	Parameter error in the Cobol program call -> specify valid OPCODE or ELEMENT-TYPE
KXL_RC_BUFFER_TOO_SMALL (51)	U	Cobol program buffer too small for a return value -> increase size of buffer
KXL_RC_INVALID_LENGTH (52)	U	The specified length value is invalid (negative) -> correct value
KXL_RC_ADD_PARAM_ EXPECTED(53)	U	An additional parameter was expected

5.3 Functions

5.3.1 Type conversion to the TVALUE structure

Function mappings

- KXLFromShort
- KXLFromInt
- KXLFromLong
- KXLFromFloat
- KXLFromDouble
- KXLFromChar
- KXLFromString
- KXLFromStruct
- KXLFromArray

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-CONVERT-FROM
	ELEMENT-TYPE	= KC-XML-SHORT / KC-XML-INT / KC-XML-LONG / KC-XML-FLOAT / KC-XML-DOUBLE / KC-XML-CHAR / KC-XML-STRING / KC-XML-STRUCT / KC-XML-ARRAY
	ELEM-VALUE-LTH	= Length of the ELEMENT-VALUE field, with KXLFromString or KXLFromStruct or = 0 with KXLFromArray
	TYPE-LTH	= Length of the TVALUE-TYPE field
	VALUE-LTH	= Length of the TVALUE-VALUE field
	VAL-xxx	= Content of the element to be converted of type xxx, where xxx= SHORT / INT / LONG / FLOAT / DOUBLE

ELEMENT-VALUE with the content of the element (string) with ELEMENT-TYPE =
KC-XML-STRING / KC-XML-STRUCT / KC-XML-ARRAY

Output parameters:
TVALUE-TYPE with the type (printable) of the element
TVALUE-VALUE with the printable value of the element

Call:
```
CALL "KXLFUNC" USING KC-XML-PARAMETER,
                     TVALUE-TYPE,
                     TVALUE-VALUE
                     (,ELEMENT-VALUE).
```

Notes:

- a short value corresponds to PIC 9(4) COMP
- an int/long value corresponds to PIC 9(8) COMP
- a float value corresponds to COMP-1 in BS2000
- a double value corresponds to COMP-2 in BS2000

5.3.2 Type conversion from the TVALUE structure

Function mappings

- KXLToShort
- KXLToInt
- KXLToLong
- KXLToFloat
- KXLToDouble
- KXLToChar
- KXLToString
- KXLToStruct

Input parameter:

```
KC-XML-PARAMETER with: OPCODE           = KC-XML-CONVERT-TO
                       ELEMENT-TYPE     = KC-XML-SHORT / KC-XML-INT /
                                          KC-XML-LONG / KC-XML-FLOAT /
                                          KC-XML-DOUBLE / KC-XML-CHAR /
                                          KC-XML-STRING / KC-XML-STRUCT
                       ELEM-VALUE-LTH   = Length of the ELEMENT-VALUE field
                       TYPE-LTH         = Length of the TVALUE-TYPE field
                       VALUE-LTH        = Length of the TVALUE-VALUE field
```

TVALUE-TYPE with the type (printable) of the element
TVALUE-VALUE with the printable value of the element

Output parameters:

```
KC-XML-PARAMETER with    VAL-xxx (where KC-XML-xxx with xxx = SHORT / INT / LONG /
                                                      FLOAT / DOUBLE )
  or
ELEMENT-VALUE with the content of the element (with KC-XML-STRING / KC-XML-STRUCT)
```

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
TVALUE-TYPE,
TVALUE-VALUE
(,ELEMENT-VALUE).

5.3.3 KXLCreateNewObj

Input parameter:

```
KC-XML-PARAMETER with: OPCODE           = KC-XML-CREATE-OBJECT
                       ELEM-NAME-LTH    = Length of the ELEMENT-NAME field
                       TYPE-LTH         = Length of the TVALUE-TYPE field
                       VALUE-LTH        = Length of the TVALUE-VALUE field
```

ELEMENT-NAME	Name of the root element
TVALUE-TYPE	(printable) type of the root element
TVALUE-VALUE	(printable) value of the root element

Output parameter:

KC-XML-PARAMETER with NODE-REFERENCE

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME,
TVALUE-TYPE,
TVALUE-VALUE.

5.3.4 KXLWrite

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-WRITE
	NODE-REFERENCE	= Reference to the current node
	ELEM-NAME-LTH	= Length of the ELEMENT-NAME field
	TYPE-LTH	= Length of the TVALUE-TYPE field
	VALUE-LTH	= Length of the TVALUE-VALUE field

ELEMENT-NAME	Name of the element
TVALUE-TYPE	(printable) type of the element
TVALUE-VALUE	(printable) value of the element

Output parameter:
KC-XML-PARAMETER with NEW-NODE-REFERENCE

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME,
TVALUE-TYPE,
TVALUE-VALUE.

Note: If TYPE-LTH=0, the TVALUE-TYPE parameter is not evaluated and no type attribute is generated.

5.3.5 KXLConvObjToDoc

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-CONVERT-OBJ-TO-DOC
	BUFFER-LTH	= Maximum length of the output buffer
	NODE-REFERENCE	= Reference to the root node of the XML object
	VALUE-LTH	= Length of the STYLESHEET-VALUE field

STYLESHEET-VALUE	Value of the stylesheet PI

Output parameter:
KC-XML-PARAMETER with BUFFER-RETURN-LTH
DOCUMENT-BUFFER

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
STYLESHEET-VALUE,
DOCUMENT-BUFFER.

5.3.6 KXLConvDocToObj

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-CONVERT-DOC-TO-OBJ
	BUFFER-LTH	= Length of the document

DOCUMENT-BUFFER	XML document

Output parameter:
KC-XML-PARAMETER with NEW-NODE-REFERENCE

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
DOCUMENT-BUFFER.

5.3.7 KXLFreeObj

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-FREE-OBJECT
	NODE-REFERENCE	= Reference to the current root node

Output parameter: no additional parameters

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER.

Note: When an object is released, all references (NODE-REFERENCE, NEW-NODE-REFERENCE) of the associated elements are invalid and may no longer be used.

5.3.8 KXLSetSubObject

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-SET-SUBOBJECT
	NODE-REFERENCE	= Reference to the current node
	ELEM-NAME-LTH	= Length of the ELEMENT-NAME field
ELEMENT-NAME	Name of the element searched for	

Output parameter:
KC-XML-PARAMETER with NEW-NODE-REFERENCE

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME.

5.3.9 KXLSetRootNode

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-SET-ROOT-NODE
	NODE-REFERENCE	= Reference to the current node

Output parameter:
KC-XML-PARAMETER with NEW-NODE-REFERENCE

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER.

5.3.10 KXLSetParentNode

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-SET-PARENT-NODE
	NODE-REFERENCE	= Reference to the current node

Output parameter:
KC-XML-PARAMETER with NEW-NODE-REFERENCE

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER.

5.3.11 KXLRead

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-READ
	NODE-REFERENCE	= Reference to the current node
	DELETION-FLAG	= KC-XML-FALSE / -TRUE
	ELEM-NAME-LTH	= Length of the ELEMENT-NAME field
	TYPE-LTH	= Length of the TVALUE-TYPE field
	VALUE-LTH	= Length of the TVALUE-VALUE field

ELEMENT-NAME	Name of the element to be read

Output parameter:

KC-XML-PARAMETER with:	TYPE-RETURN-LTH	= Return length of the TVALUE-TYPE field
TVALUE-TYPE	(printable) type of the element	
TVALUE-VALUE	(printable) value of the element	

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME,
TVALUE-TYPE,
TVALUE-VALUE.

Note: If the read element contains no type attribute, TYPE-RETURN-LTH=0 is returned. TVALUE-TYPE is then not modified.

5.3.12 KXLReadNode

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-READ-NODE
	NODE-REFERENCE	= Reference to the current node
	ELEM-NAME-LTH	= Length of the ELEMENT-NAME field
	TYPE-LTH	= Length of the TVALUE-TYPE field
	VALUE-LTH	= Length of the TVALUE-VALUE field
	FULL-NAME	= KC-XML-FULL-NAME/ KC-XML-PARTIAL-NAME

Output parameter:

KC-XML-PARAMETER with:	TYPE-RETURN-LTH	= Return length of the TVALUE-TYPE field
ELEMENT-NAME	Name of the node read	
TVALUE-TYPE	(printable) type of the node	
TVALUE-VALUE	(printable) value of the node	

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME,
TVALUE-TYPE,
TVALUE-VALUE.

Note: If the read element contains no type attribute, TYPE-RETURN-LTH=0 is returned. TVALUE-TYPE is then not modified.

5.3.13 KXLReadNextSib

Input parameter:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-READ-NEXT-SIB
	NODE-REFERENCE	= Reference to the current node
	ELEM-NAME-LTH	= Length of the ELEMENT-NAME field
	TYPE-LTH	= Length of the TVALUE-TYPE field
	VALUE-LTH	= Length of the TVALUE-VALUE field

Output parameter:

KC-XML-PARAMETER with	NEW-NODE-REFERENCE
	TYPE-RETURN-LTH
ELEMENT-NAME	Name of the node read
TVALUE-TYPE	(printable) type of the node
TVALUE-VALUE	(printable) value of the node

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME,
TVALUE-TYPE,
TVALUE-VALUE.

Note: If the read element contains no type attribute, TYPE-RETURN-LTH=0 is returned. TVALUE-TYPE is then not modified.

5.3.14 KXLReadChild

Similar to KXLReadNextSib with
OPCODE = KC-XML-READ-CHILD as input parameter.

5.3.15 KXLReadNextSingleNode

Similar to KXLReadNextSib with
OPCODE = KC-XML-READ-NEXT-SINGLE-NODE as input parameter.

5.3.16 KXLReadAttr

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-READ-ATTR
	NODE-REFERENCE	= Reference to the current node
	ELEM-NAME-LTH	= Length of the field ELEMENT-NAME
	VALUE-LTH	= Length of the field TVALUE-VALUE

Output parameters:

KC-XML-PARAMETER with	NEW-NODE-REFERENCE
ELEMENT-NAME	Name of the attribute read
TVALUE-VALUE	(printable) value of the attribute

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME,
TVALUE-VALUE.

5.3.17 KXLFindNode

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-FIND-NODE
	NODE-REFERENCE	= Pointer to the current element
	ELEM-NAME-LTH	= Length of the field ELEMENT-NAME
	TYPE-LTH	= Length of the field TVALUE-TYPE
	VALUE-LTH	= Length of the field TVALUE-VALUE
	PREFIX2-LTH	= Length of the field PREFIX
	URL2-LTH	= Length of the field URL

ELEMENT-NAME Name of the element

Output parameters:

KC-XML-PARAMETER with	NEW-NODE-REFERENCE TYPE-RETURN-LTH = length of the TVALUE-TYPE field
TVALUE-TYPE	(printable) type of the element
TVALUE-VALUE	(printable) value of the element
PREFIX	Namespace prefix or SPACES, if the newly found element is assigned to the default namespace or no namespace at all.
URL	Namespace URL or SPACES, if the newly found element is not assigned to any namespace.

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME,
TVALUE-TYPE,
TVALUE-VALUE,
PREFIX,
URL.

Note: If the read element contains no type attribute, TYPE-RETURN-LTH=0 is returned. TVALUE-TYPE is then not modified.

5.3.18 KXLGetSizeofNodelist

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-GET-SIZEOF-NODELIST
	ELEM-NAME-LTH	= Length of the field ELEMENT-NAME
	NODE-REFERENCE	= Pointer to the current element

ELEMENT-NAME Name of the element

Output parameter:
KC-XML-PARAMETER with NODELIST-LENGTH

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ELEMENT-NAME.

5.3.19 KXLGetHomeEnc

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-GET-HOME-ENC
	ELEM-NAME-LTH	= Length of the field ENCODING-NAME

Out:
ENCODING-NAME Name of the home encoding

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ENCODING-NAME.

5.3.20 KXLGetDocEnc

Input parameters:

```
KC-XML-PARAMETER with:  OPCODE          = KC-XML-GET-DOC-ENC
                        NODE-REFERENCE  = Reference to the current node
                        ELEM-NAME-LTH   = Length of the field ENCODING-NAME
```

Output parameter:

ENCODING-NAME Name of the document encoding

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ENCODING-NAME.

5.3.21 KXLSetDocEnc

Input parameters:

```
KC-XML-PARAMETER with:  OPCODE          = KC-XML-SET-DOC-ENC
                        NODE-REFERENCE  = Reference to the current node
                        ELEM-NAME-LTH   = Length of the field ENCODING-NAME
```

ENCODING-NAME Name of the document encoding

Output parameter: no additional parameters

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ENCODING-NAME.

5.3.22 KXLStringToUTF8

Input parameters:

```
KC-XML-PARAMETER with:  OPCODE             = KC-XML-STRING-TO-UTF8
                        BUFFER-LTH         = Length of the field INPUT-STRING
                        BUFFER-RETURN-LTH  = Length of the field OUTPUT-STRING
```

INPUT-STRING string to be converted

Output parameter: OUTPUT-STRING

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
INPUT-STRING,
OUTPUT-STRING.

5.3.23 KXLStringFromUTF8

Input parameters:

```
KC-XML-PARAMETER with:  OPCODE             = KC-XML-STRING-FROM-UTF8
                        BUFFER-LTH         = Length of the field INPUT-STRING
                        BUFFER-RETURN-LTH  = Length of the field OUTPUT-STRING
```

INPUT-STRING string to be converted

Output parameter: OUTPUT-STRING

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
INPUT-STRING,
OUTPUT-STRING.

5.3.24 KXLGetEncodingAlias

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-GET-ENCODING-ALIAS
	BUF1-LTH	= Length of the field ALIAS-NAME
	BUF2-LTH	= Length of the field ENCODING-NAME

ALIAS-NAME	alias name

Output parameters:

ENCODING-NAME	encoding name

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ALIAS-NAME,
ENCODING-NAME.

5.3.25 KXLSetEncodingAlias

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-SET-ENCODING-ALIAS
	BUF1-LTH	= Length of the field ALIAS-NAME
	BUF2-LTH	= Length of the field ENCODING-NAME

ALIAS-NAME	alias name
ENCODING-NAME	encoding name

Output parameters: no additional

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ALIAS-NAME,
ENCODING-NAME.

Note: If the current assignment of the alias name should be removed BUF2-LTH must be set to 0.

5.3.26 KXLWriteNS

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-WRITE-NAMESPACE
	NODE-REFERENCE	= Reference to the current node
	PREFIX-LTH	= Length of the field PREFIX
	URL-LTH	= Length of the field URL

PREFIX	Prefix of the namespace
URL	url of the namespace

Output parameter: no additional

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
PREFIX,
URL.

Note: PREFIX-LTH is limited to 16 and URL-LTH is limited to 128, otherwise the entry from the call KXLReadNSList cannot be read.

5.3.27 KXLDelNS

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-DELETE-NAMESPACE
	NODE-REFERENCE	= Reference to the current node
	PREFIX-LTH	= Length of the field PREFIX

PREFIX	prefix of the namespace

Output parameters: no additional

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
PREFIX.

5.3.28 KXLReadNSList

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-READ-NAMESPACE-LIST
	NODE-REFERENCE	= Reference to the current node
	BUFFER-LTH	= Length of the field NS-TBL-BUFFER

Output parameter:
KC-XML-PARAMETER with BUFFER-RETURN-LTH
NS-TBL-BUFFER

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
NS-TBL-BUFFER.

Structure of the table, which is returned in NS-TBL-BUFFER:

```
41 NS-TBL-ENTRIES.
  43 NS-TBL-LTH         PIC 9(8) COMP-5.
  43 NS-ENTRY           OCCURS 0 TO 31 DEPENDING ON NS-TBL-LTH.
    45 NS-PREFIX        PIC X(16).
    45 NS-URL           PIC X(128).
```

Note: Apart from the C interface, character strings are delivered in home encoding.

5.3.29 KXLSearchNS

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-SEARCH-NAMESPACE
	NODE-REFERENCE	= Reference to the current node
	PREFIX-LTH	= Length of the field PREFIX
	URL-LTH	= Length of the field URL

PREFIX	prefix of the namespace or
URL	url of the namespace

Output parameter: PREFIX, URL

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
PREFIX,
URL.

Note: Apart from the C interface, character strings are delivered in home encoding.

5.3.30 KXLConvDocToObjAndValid

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-CONVERT-DOC-AND-VALID
	BUFFER-LTH	= Length of the field DOC-BUFFER
	BUF2-LTH	= Length of the field LOCAL-DIR
	VALIDATION-FLAG	= KC-XML-FALSE / -TRUE

DOC-BUFFER	XML document
LOCAL-DIR	local directory

Output parameters:
KC-XML-PARAMETER with NEW-NODE-REFERENCE

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
DOC-BUFFER,
LOCAL-DIR.

Note:
- The LOCAL-DIR parameter matches the pDir parameter of the C API.
- If you do not want to work with local files, then the LOCAL-DIR parameter must contain LOW-VALUE.

5.3.31 KXLParseSchema

Input parameters:

KC-XML-PARAMETER with:	OPCODE	= KC-XML-SCHEMA-PARSE-BUF
	BUFFER-LTH	= Length of the field SCHEMA-BUFFER
	BUF2-LTH	= Length of the field LOCAL-DIR

SCHEMA-BUFFER	XML schema
LOCAL-DIR	local directory

Output parameters: SCHEMA-REFERENCE in KC-XML-PARAMETER

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
SCHEMA-BUFFER,
LOCAL-DIR.

Note:
- The LOCAL-DIR parameter matches the pDir parameter of the C-API.
- If you do not want to work with local files, then the LOCAL-DIR parameter must contain LOW-VALUE.

5.3.32 KXLParseSchemaFile

Input parameters:

```
KC-XML-PARAMETER with: OPCODE           = KC-XML-SCHEMA-PARSE-FILE
                       URL-LTH          = Length of the file name FILENAME
                       BUF2-LTH         = Length of the field LOCAL-DIR
```

FILENAME	name of the file containing the XML schema
LOCAL-DIR	local directory

Output parameters: SCHEMA-REFERENCE in KC-XML-PARAMETER

Call:
```
CALL "KXLFUNC" USING KC-XML-PARAMETER,
                     FILENAME,
                     LOCAL-DIR.
```

Note:
- The LOCAL-DIR parameter matches the pDir parameter of the C-API.
- If you do not want to work with local files, then the LOCAL-DIR parameter must contain LOW-VALUE.

5.3.33 KXLValidDocBuf

Input parameters:

```
KC-XML-PARAMETER with: OPCODE            = KC-XML-SCHEMA-VALIDATE-DOC
                       BUFFER-LTH        = Length of the field DOC-BUFFER
                       SCHEMA-REFERENCE  = Pointer to the XML schema structure
```

DOC-BUFFER	XML document

Output parameters: none

Call:
```
CALL "KXLFUNC" USING KC-XML-PARAMETER,
                     DOC-BUFFER.
```

5.3.34 KXLValidDoc

Input parameters:

```
KC-XML-PARAMETER with: OPCODE            = KC-XML-SCHEMA-VALIDATE-OBJ
                       NODE-REFERENCE    = Reference to the root node of the
                                           document
                       SCHEMA-REFERENCE  = Pointer to the XML schema structure
```

Output parameters: none

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER.

5.3.35 KXLFreeSchema

Input parameters:

```
KC-XML-PARAMETER with: OPCODE           = KC-XML-SCHEMA-FREE
                              SCHEMA-REFERENCE = Pointer to the XML schema structure
```

Output parameters: none

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER.

5.3.36 KXLTSENV

Input parameters:

```
KC-XML-PARAMETER with: OPCODE               = KC-XML-INIT-TRACE
```

Output parameters: none

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER.

5.3.37 KXLInitEnv

Input parameters:

```
KC-XML-PARAMETER with: OPCODE               = KC-XML-INIT-ENV
                       ENC-FUNC-TO-UTF8     = Address of the encoding function from
                                              home encoding to UTF-8
                       ENC-FUNC-FROM-UTF8 = Address of the encoding function from
                                              UTF-8 to home encoding
                       BUFFER-LTH           = Length of the field ENCODING-NAME

ENCODING-NAME          name of the home encoding
```

Output parameters: none

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ENCODING-NAME.

Caution:

1. The home encoding name must be specified in UTF-8.
2. Notice, that the given encoding functions (must) process and create C strings (ending with '/0'), because these functions are called in UTM-XML from C functions.

5.3.38 KXLSchemaGetRoot (requesting the root node of a schema object)

Input parameters:

```
KC-XML-PARAMETER with: OPCODE               = KC-XML-SCHEMA-GET-ROOT
                       SCHEMA-REFERENCE = Address of the XML schema structure
```

Output parameters:

```
KC-XML-PARAMETER with  NEW-NODE-REFERENCE      root node of the schema object
```

Return code: If SCHEMA-REFERENCE is a NULL pointer, then
KC-XML-RC-INVALID-PARAMETER is returned.

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER.

Function:
This function expects a pointer to a schema structure and returns the root pointer of the corresponding schema object. With this pointer you can create the schema document via KXLConvObjToDoc.
This function is only available in the COBOL interface.

Caution: The schema object must not be freed with KXLFreeObj. If you do not need the schema structure any more, then you should free it with KXLFreeSchema.

Return codes:

Name (value)	Error Type	Comment-> Action
KXL_RC_OK (0)	I	Function carried out successfully
KC-XML-RC-INVALID-PARAMETER (32)	U	The address of the XML schema structure is NULL

5.3.39 KXLGetLastParserError

Input parameters:

KC-XML-PARAMETER with: OPCODE	= KC-XML-GET-PARSER-ERRMSG
BUFFER-LTH	= Length of the field ERR-MESSAGE

Output parameters:
KC-XML-PARAMETER with PARSER-RTCODE
ERR-MESSAGE

Call: CALL "KXLFUNC" USING KC-XML-PARAMETER,
ERR-MESSAGE.

6 Installation

UTM XML is only available via download from the Internet. You can download platform-specific application packages ('rt') and source packages ('dev') of UTM XML from the following Web page:

http://ts.fujitsu.com/products/software/openseas/openutm.html

After downloading, you may need to transfer the packages to the target platforms. The packages are compressed and must be decompressed using the proper tools (tar, winzip).

The application package contains all the components you need in addition to UTM to create the UTM XML program units:

- Documentation: License, Readme files, function description (this document)
- COBOL COPY element
- C header files
- source code of the UTM sample programs in Cobol and C
- Libraries or BS2000 load modules

The source code package contains the UTM XML source code, the GNOME Parser libxml2 source code, the license, and the descriptions.

To use UTM XML, you only need to install the components of the application package.

6.1.1 Unix platforms

There are different packages available on the Web page stated above for the different platforms and variants (32-bit/64-bit). The name of the package shows which target platform and variant a package is used for. For example, "rt" designates the UTM XML application package including all source code for the sample programs in COBOL and C. The packages with "dev" in their name contain all source code with header files and examples.

An exact list of the packages available can be found in the Release Notice (Readme).

On UNIX and Linux platforms, you may have to transfer the packages to the target machine after they have been downloaded.

After this, the packages with suffix .Z must be uncompressed with uncompress and the packages with suffix .gz must be uncompressed with gunzip. Packages with suffix .tar must be unpacked with tar -xvf..

6.1.2 BS2000

The packages are available in a ZIP archive for BS2000. After downloading, unpack the ZIP file, which contains the following subdirectories:

- ftp with the LMS library SYSLIB.UTM-XML.<version>.RT or -.DEV
- openft with the LMS library SYSLIB.UTM-XML.<version>.RT or -.DEV
- doc with the documents (Readme, API description, license)

Transfer the LMS library from the ftp subdirectory to your BS2000 system if you perform the transfer with ftp (in the binary mode). Or you can transfer the LMS library from the openft subdirectory to your BS2000 system if you perform the transfer with openFT (binary file type, transparent transfer mode).

After the transfer they are available and ready for use as LMS libraries.

The following list provides an overview of the contents of the LMS libraries:

SYSLIB.UTM-XML.<version>.RT (UTM XML application library) contains

- the XML objects for BS2000/OSD
- the COBOL COPY element
- the C header files
- the UTM sample programs in Cobol and C (sources and modules)
- the documentation

SYSLIB.UTM-XML.<version>.DEV (UTM XML source library) contains

- all libxml2 parser sources used with C header files
- all UTM XML source modules with C header files

6.1.3 Windows

The following packages are available for Windows on the Web page stated above:

- utmxmlrt_.<version>._win_32.zip: the UTM XML application package including the sample programs in COBOL and C.
- utmxmldev_.<version>._win_32.zip: UTM XML source package

After downloading, you may need to transfer the packages to the target machine. Unpack them then with Winzip. The components are copied to the appropriate directories with the install.cmd script. To do this for the application package, the environment variable $UTMPATH must be set to the current UTM directory. Once the ZIP file is unpacked and the corresponding install.cmd script is started, a file tree named xml is created with the corresponding subdirectories in the UTM directory. The separate components are then installed in these directories. For the source package, calling install.cmd copies the components to a file tree named xml/opensource in the current directory.

7 Usage

7.1.1 Unix platforms

When compiling components that call the XML interface, you must specify the directories <utmxml_home>/include and (for Cobol) <utmxml_home>/copy-cobol85 or <utmxml_home>/netcobol. Here <utmxml_home> is the directory in which the UTM XML application package was installed.

COBOL program units that use the COPY element KXLCOBOL must be compiled without the NOMF option when using the MicroFocus compiler, because otherwise compiler errors will occur.

When using a different compiler, the VAL-FLOAT and VAL-DOUBLE fields should be modified accordingly. If there are no comparable types for FLOAT and DOUBLE, then the fields should be set as follows:

```
43 VAL-FLOAT        PIC X(4).
43 VAL-DOUBLE       PIC X(8).
```

The corresponding functions, KXLFromFloat, KXLFromDouble, KXLToFloat, and KXLToDouble, cannot be used in COBOL in this case.
When linking a UTM application (client/server) using the UTM XML interface, the following libraries (from <utmxml_home>\lib) must be taken into account:

- libutmxml.a (static) and
- libutmxml.so/sl (dynamic)

In addition, the library containing the mathematical functions must be linked by specifying -lm.

The exact implementation can be found in the sample application (included with openUTM).

For usage of user specific encoding functions or the adaptation of existing encoding functions see Appendix, section “Usage of user specific encoding functions”.

7.1.2 BS2000

Compiling:

For calling the UTM-XML interface include the header file `libxml/kxlinc.h` in the source program. When compiling the programs that contain interface calls, assign the SYSLIB.UTM-XML.<version>.RT library with the USER-INCLUDE-LIBRARY option. C program units containing calls to the UTM XML API must be compiled and linked as LLMs.
The name of the header files libxml/<includename> correspond to the include files LIBXML.<includename> in BS2000. The BS2000 C compiler converts the names accordingly.

The module KXLCVLT.C contains the encoding tables from UTF-8 to the EBCDIC character sets and vice versa. For minimal changes of these tables you can adapt the tables in the source code (contained in SYSLIB.UTM-XML.<version>.DEV). Then you must compile the source with the C compiler using the following compiler option:

```
MODIFY-MODULE-PROPERTIES LOWER-CASE-NAMES=*YES
```

Compiling of Cobol programs:

COBOL program units that contain UTM-XML interface calls must contain the copy element KXLCOBOL (COPY instruction). When compiling the program unit you must assign the library SYSLIB.UTM-XML.<version>.RT to one of the link names COBLIB, COBLIBn (n=1,…,9). The following compiler option must be specified:

```
P[ERMIT]-S[TANDARD]-D[EVIATION]=YES
```

Linking:

If you want to link a program unit that contains calls to the XML interface to your UTM application, then you must insert a RESOLVE statement for SYSLIB.UTM-XML.<version>.RT.

If the UTM application is generated based on the BLS interface, then a BLSLIBnn link name for SYSLIB.UTM-XML.<version>.RT can be assigned in the start procedure.

If you have modified the source KXLCVLT.C you must link the compiled module explicitly.

For the usage of user specific encoding functions or the adaptation of predefined encoding functions see Appendix, section "Usage of user specific encoding functions".

7.1.3 Windows

When compiling components that call the UTM XML interface, you must take the <utmxml_home>\include and (for Cobol) <utmxml_home>\copy-cobol85 or <utmxml_home>/netcobol directories into account. <utmxml_home> is the directory in which the UTM XML application package was installed.

When linking a UTM application (client/server) using the UTM XML interface, the following libraries (from <utmxml_home>\dll) must be taken into account:

- libutmxml.lib and
- libutmxml.dll

Furthermore, the following object must be linked (from <utmxml_home>\obj) when using COBOL:

- kxlcob2c.obj

The exact implementation can be found in the Quickstart Kit (included with openUTM).

8 Diagnostics

8.1 Trace

Traces can be turned on for the XML interface to enable diagnostics to be performed while an application is running. They are controlled via environment or job variables, which must be set before the start of the application.
Process (on Unix and Windows systems) or **task** (in BS2000) is used below to designate a UTM, UPIC or other UTM client process.
Each process or task writes the trace into a separate file, two generations of which can exist (old and new). The maximum size of a trace file is about 2000 KB. As soon as this size is reached, the trace switches over to a second file. Once this has also reached the limit, the trace writes to the first file again.

The name of a trace file is made up as follows:
KXL*pid*.*n* (UNIX, Windows systems) or
KXL*tsn*.*n* (BS2000/OSD), where

KXL	denotes an XML trace,
pid	is the process ID of the process, 5-digit,
tsn	is the task ID, 4-digit,
n	is the generation number: 1 or 2.

You can recognize the more recent trace by the timestamp in the trace data.
Example: KXL00341.1: XML trace file number 1 for process 00341
KXL00341.2: XML trace file number 2 for process 00341

The function KXLTSENV() is called to initialize the trace functions of the UTM-XML interface. The call is issued implicitly with KXLInitEnv and each KXLCreateNewObj and KXLConvDocToObj, i.e. each time a new object is created. It can also be issued explicitly, however. In this case the header file `kxltrace.h` must be specified:

```
#include <libxml/kxltrace.h>
```

The trace is initialized on a process- or task-specific basis, i.e. each process/task must issue a trace call before the trace writes XML trace records.
At initialization time, the desired trace mode is queried from the environment variable or job variable ((link)name KXLTRAC) via the macro getenv (or GETJV in BS2000) and set, the name of the trace file formed and the file opened.

Note for BS2000:
In BS2000, the job variable with the link name KXLTRAC is re-read each time a KXLTSENV call is issued. If the trace mode has been modified in the meantime, the new mode is registered in the program.

The following 3 trace modes can be specified:

E (Error): Activates the error trace. After each return code not equal to KXL_RC_OK, an error message is written to the trace file:
<timestamp> <function> error: <error text>

I (interface): Activates the interface trace for the UTMXML calls. The following record is written to the trace file each time an UTMXML function is called:
<timestamp> <function> init < input parameter>
The following record is written to the trace file each time a return is made from an UTMXML function:
<timestamp> <function> exit <return parameter>
As the trace modes are based on a hierarchical structure, trace mode I also includes trace mode E.

F (Full): Activates the full XML trace. In addition to the scope of trace mode I, the XML document is output to the trace file each time a conversion is performed. If the document is greater than 4096 characters, only 2048 bytes are output from the beginning of the document, and 2048 bytes from the end of the document.

The input and return parameters of the relevant function are logged in the following form under <input parameter> and <return parameter> and separated from one another by commas:

xmlNodePtr	Address in the form	XML-A=nnnnnnnn
char * (docu.)	Address in the form	DOC-A=nnnnnnnn
xmlSchemaPtr	Address in the form	SCA-A=nnnnnnnn
char * (stylesheet)	Character string in the form	S=<string>
char * (Name)	Character string in the form	N=<string>
t_value	2 character strings in the form	T=<string>, V=<string>
short (RetCode)	Numeric value of the return code	RC=nn
int (Length)	Integer length value	LTH=nn
int (NameType)	Numeric value of the element type	ET=nn
int (FullName)	Boolean value for name editing	FULL=0/1
int (Delete)	Boolean value for deletion	DEL=0/1
short/int/long/ float/double/char *	Input/return of the type conversion functions	SH/IN/LG/FL/DO/ST=<value>
char *(Name)	Encoding name	ENC=<string>
Ptr	Encoding address	ENC-A= nnnnnnn
int	Parser return code	P-RC=nn
char *(message)	Parser error message	MSG=<string>
int	Size of Nodelist	SIZ=nn
int	Version	V=nn
int (Prefix length)	Prefix length value	P_LTH=nn
char *(Präfix)	Namespace Präfix	NS-P=<string>
char *(Url)	Namespace URL	NS-U=<string>

If FL/DO = <value> is specified, the hexadecimal value of the variable is logged. The following line (printable repetition) is not important.

If output parameters are not logged, e.g. in the case of an error, the values are as follows:
- for addresses, NULL,
- for names, empty string '\0',
- for t_value {NULL, NULL}.

The meaning of the values of ET is as follows:
0 NoType
1 IsSingleType
2 IsStruct
3 IsArray
4 IsElemOfArray

The meanings of the values of P-RC are described in xmlParserErrors in the parser header file xmlerror.h. If P-RC = -1, then there is an internal parser error or a parameter error.

A sample trace file is shown in the example in the Appendix.

Comment:
If an error is detected in the parser, a message is given. The message is output to the current trace file, if the trace file is installed, or to SYSOUT/stderr.

8.1.1 Store trace mode on UNIX and Windows systems

For the UTM-XML API, the trace mode is stored in the environment variable KXLTRAC. For possible trace mode options (E/I/F), see above. Each process reads this environment variable in the first call of KXLTSENV and initializes the trace with the desired trace mode.
Environment variables are set on UNIX systems with the following command:

```
KXLTRAC = value
export KXLTRAC
```

On UNIX systems, the environment variables apply to just one shell at a time; other values may be valid for an application in another shell.
Under Windows, environment variables can be set with the SET command before the start of the program.
It is recommended that environment variables be defined via Desktop – Properties - Environment.

8.1.2 Store trace mode under BS2000/OSD

For the XML API, the desired trace mode is to be stored in a job variable with the link name KXLTRAC. When KXLTSENV is called, each task reads this job variable and initializes the trace with the desired trace mode.

If the JV software product is loaded as a subsystem, the job variables can be set as follows under BS2000/OSD V3.0, for example:

1. Create job variable:
   CREATE-JV JV-NAME=FULLTR
2. Pass value to the job variable:
   MODIFY-JV JV[-CONTENTS]=FULLTR, SET-VALUE='F'
3. Set task-specific job variable link:
   SET-JV-LINK LINK-NAME=KXLTRAC, JV-NAME=FULLTR
4. Show task-specific job variable link:
   SHOW-JV-LINK JV[-NAME]=FULLTR
5. Delete task-specific job variable link:
   REMOVE-JV-LINK LINK-NAME=KXLTRAC

Under BS2000/OSD, the job variable links are task-specific. Other job variables can be assigned to a second application under the same user ID.

9 Special points when using the API under openUTM

The basic principle of the DOM interface is the following:
An object with a tree structure is built in the program storage space. The object can be processed interactively and it is possible to position within the object. In other words, starting from an interior node, it is possible to work on a subtree without needing to pass through the full path from the root to the node each time a node is accessed (enhanced performance!).
The object is converted into a document for sending to communication partners or for archiving.
When the UTM-XML API is used in openUTM, i.e. in a UTM program unit, it is not possible to access the object and any positionings present after the end of the program unit due to a possible process change.
For this reason there are two options for interactive processing of a document:

1. The XML document is stored in UTM storage areas, e.g. in the KB or GSSB. Each time a program unit is started (=start of dialog step), the document is read and converted into an object; after processing and before the end of the program unit (=end of dialog step), the object is converted back again and stored/sent as a document. This option is inefficient, particularly with large documents.
2. A program unit is used with PGWT. The object is retained in the storage space even at the end of the dialog step (MPUT, PGWT, MGET). In other words, it is possible to continue working on the present object after an output on or input from the screen.
   Disadvantages:
   - There is no backup, i.e. UTM-side transaction security (e.g. document backup to KB or GSSB) becomes effective at the end of the program unit at the earliest. Then, however, the object is no longer available in the storage space (and should therefore be released before the end of the program unit).
   - In each case, a UTM process is 'blocked' by each PGWT program unit, i.e. even after the end of the dialog step, the process is not released for other users. In the worst case, therefore, there must be as many processes as users.

When the UTM-XML API is used in openUTM clients, such as UPIC, the problem does not arise, as the context is preserved across dialog step boundaries.

10 Appendix

10.1 Return codes

The values of the return code returned by the functions of this interface are defined with #define in the include file `kxlinc.h`, i.e. they are int values by default. To simplify diagnosis when errors occur, the return code can be output as a character string in C/C++ programs. By defining the symbolic name KXL_GEN_STRING before including `kxlinc.h`:

```
  #define KXL_GEN_STRING
  #include kxlinc.h
```

it is possible to access the desired character string (max. 32 characters) with KXL_RCODE_STRINGS[returncode].

The values KC-XML-RC-xxx of RTCODE in COBOL correspond to the values KXL_RC_xxx in `kxlinc.h` in C.
Name, value and meaning of the return codes are shown below. The “Error type” column can contain: U (User error), S (System error) or I (Information).

Return codes:

Name (value)	Error type	Comment -> Action
KXL_RC_OK (0)	I	Function executed correctly
KXL_RC_NO_NODE_FOUND (1)	I	Node with the specified name not found
KXL_RC_DIFF_TYPES(2)	I	type in t_value and requested type are different
KXL_RC_NO_ROOT_CREATED (4)	S	The parser could not create a new object; ->see parser message (trace file or KXLGetLastParserError).
KXL_RC_NO_CONVERSION_TO_DOC (7)	S	Error when creating the document -> analyze parser message (trace file or KXLGetLastParserError)
KXL_RC_EMPTY_DOC (9)	U	Converted object contains no nodes
KXL_RC_INVALID_DOC_PTR (10)	S/U	The specified object node is not assigned to any object Possible cause: object destroyed or created incorrectly -> generate error documentation
KXL_RC_ALLOC_ERROR (11)	S	No more new storage space could be requested with malloc/realloc
KXL_RC_TYPE_MISMATCH(12)	U	A node is to be overwritten, but the existing type does not match the specified type -> if you want to overwrite the node, first delete it, then write a new one
KXL_RC_NO_CHILD_CREATED (13)	S/U	Error during creation of a new node -> analyze parser message
KXL_RC_NO_TYPE_ATTR_ FOUND(14)	S/U	When reading/overwriting a node, no type attribute was found; no positioning/overwriting possible.
KXL_RC_NO_XML_NODE (15)	U	The pointer to the current node is NULL -> specify valid pointer

Name (value)	Error type	Comment -> Action
KXL_RC_NO_SUBTREE_ DELETION (16)	U	When reading a node with delete, the node was not deleted because it is not an leaf node. -> if the entire subtree is to be deleted, each node must be deleted individually.
KXL_RC_INVALID_T_VALUE (17)	U	One of the pointers specified in t_value is NULL -> correct value
KXL_RC_EMPTY_NAME(18)	U	When analyzing the element name, an empty name part was found; writing aborted or name empty -> specify valid name.
KXL_RC_INVALID_NAME (19)	U	When analyzing the element name, a syntax error was detected in the use of "[", "]" , "/" and @ -> specify valid name
KXL_RC_PARSER_ERROR (20)	S/U	Error found in parser. Parser RC or message can be recovered with KXLGetLastParserError.
KXL_RC_NO_CONTEXT_FOR_ PARSER (21)	S	Error in setting parser internal management areas. -> solve possible memory bottleneck
KXL_RC_CONVERSION_ERROR (22)	S/U	Error in code conversion
KXL_RC_VERSION_ERR (23)	U	Inconsistent UTM XML module versions.
KXL_RC_PARSER_VERS_NOT_ SUPP (24)	U	Parser version not supported
KXL_RC_ATTR_NAME_ERROR (25)	U	Incorrect attribute name structure; no '@' is allowed after '/', '[' or ']'.
KXL_RC_ATTR_READ_WRITE_ ERR (26)	U	Read or write error of an attribute
KXL_RC_ENC_NAME_ERR (27)	U	Encoding name too long (> 15 characters)
KXL_RC_NODE_TYPE_NOT_ ALLOWED (28)	U	This call can only be used with an element node or attribute node.
KXL_RC_NO_PARENT_NODE (29)	U	Specified node has no reference to a parent node.
KXL_RC_UNKNOWN_ENCODING (30)	U	Encoding name is unknown to the parser. - > declare encoding conversion routines to the parser
KXL_RC_NAMESPACE_NOT_ FOUND (31)	I	Namespace not found
KXL_RC_INVALID_PARAMETER (32)	U	Incorrect parameter specification, e.g. NULL pointer specified, although not allowed.
KXL_RC_NAMESPACE_WRITE_ ERROR (33)	S/U	The parser could not write the namespace definition correctly, see parser error messages.
KXL_RC_NAMESPACE_DEL_ERR (34)	U	The namespace definition could not be deleted. Elements are still assigned to it.
KXL_RC_ ELEM_BEL_TO_NAMSP (35)	U	At least one element is assigned to the current namespace.
KXL_RC_ENCODING_CHANGE_ ERR (36)	U/S	No encoding handler could be set up for the specified home encoding
KXL_RC_PARSER_CONTEXT_ ERR (37)	S	Internal parser error while creating internal parser management spaces -> eliminate any eventual memory bottlenecks
KXL_RC_SCHEMA_PARSE_INT_ ERR (38)	U	Error while parsing the schema -> see the message from the parser. (trace file or KXLGetLastParserError)
KXL_RC_SCHEMA_VALID_INT_ ERR (39)	U	Error during validation -> see the message from the parser. (trace file or KXLGetLastParserError)

Name (value)	Error type	Comment -> Action
KXL_RC_NO_SCHEMA_GIVEN(40)	U	The schema input address is NULL
KXL_RC_FILE_OPEN_ERR (41)	S	Error while opening a file
KXL_RC_FILE_READ_ERR (42)	S	Error while reading a file
KXL_RC_FILE_CLOSE_ERR (43)	S	Error while closing a file
KXL_RC_DOC_NOT_VALID(44)	U	The document is not a valid instance of the specified schema -> see the message from the parser. (trace file or KXLGetLastParserError)
KXL_RC_IO_HNDLR_INIT_ERR (45)	U	To many IO handlers defined
KXL_RC_PARSER_ENCODING_ERR (46)	S/U	Parser couldn’t execute the function -> check if alias name and, if needed, encoding name are defined correctly.
KXL_RC_IS_ALIAS_ENCODING (47)	U	Given encoding name is alias name -> specify original encoding name
KXL_RC_COBOL_PARAM_ERROR(50)	U	Parameter error in the Cobol program call -> specify valid OPCODE or ELEMENT-TYPE
KXL_RC_BUFFER_TOO_SMALL (51)	U	Cobol program buffer too small for a return value -> increase size of buffer
KXL_RC_INVALID_LENGTH(52)	U	The specified length value is invalid (negative or too great) -> correct specification
KXL_RC_NO_PARENT_GIVEN(90)	S/U	When a call is issued to KXLWrite (internal function xmlNewChild), the reference to parent is NULL
KXL_RC_NO_NODE_NAME_GIVEN(91)	U	When a call is issued to KXLWrite (internal function xmlNewChild), the name is not specified
KXL_RC_NOT_SUPPORTED(99)	U	Function not (yet) supported

The following return code values are not returned, but are defined (as upper and lower limit):

KXL_RC_NIL (-1)	not valid
KXL_MAX_RC(99)	highest return code

The following return codes are not returned, but are written to the trace file by the file handling functions when an error occurs:

KXL_RC_FILE_OPEN_ERR (41)	Error while opening a file
KXL_RC_FILE_READ_ERR (42)	Error while reading a file
KXL_RC_FILE_CLOSE_ERR (43)	Error while closing a file

When errors are detected by the parser (e.g. XML syntax errors in the document), a message is output to stderr.

10.2 Usage of user specific encoding functions

For the handling of user specific encoding functions there are several possibilities:

1. You can develop your own functions according to the prototypes (see section "Initialization of the UTM XML interface"), which manage the code conversion. You must specify them on the first call of KXLInitEnv() or announce them directly to the parser with xmlNewEncodingHandler().
2. You can adapt an existing code table (in source KXLCVLT.C). After compiling you must link the module to your application. The changes affect the corresponding encoding (Not possible on Windows!).
3. You can build new code tables according to the structure of the existing tables. In addition you build two functions according to the prototypes (see section 4.1 "Initialization of the UTM XML interface"). They have to call functions of UTM-XML with the newly built code tables. These functions must be announced to the parser such as in possibility 1. These functions can be built as well easily with means of Cobol (see example KXLCOBST.CBL).

For option 2. and 3. in the following sections the structure of the tables is described.

10.2.1 Code table UserEncoding -> UTF-8

The following example of the EDF03DRV table shows the structure of the code table transforming user encoding to UTF-8. For each character in EDF03DRV with binary value i the hexadecimal UTF-8 value is given at position i (0 .. 255) of the table. Characters which are not defined are transformed to 0x1a.

```
unsigned char xmlunicodetable_EDF03DRV[256] = {
         0x00, 0x01, 0x02, 0x03, 0x1a, 0x09, 0x1a, 0x7f, /* X'00' - X'07' */
         0x1a, 0x1a, 0x1a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, /* X'08' - X'0F' */
         0x10, 0x11, 0x12, 0x13, 0x1a, 0x1a, 0x08, 0x1a, /* X'10' - X'17' */
         0x18, 0x19, 0x1a, 0x1a, 0x1c, 0x1d, 0x1e, 0x1f, /* X'18' - X'1F' */
         0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x0a, 0x17, 0x1b, /* X'20' - X'27' */
         0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x05, 0x06, 0x07, /* X'28' - X'2F' */
         0x1a, 0x1a, 0x16, 0x1a, 0x1a, 0x1a, 0x1a, 0x04, /* X'30' - X'37' */
         0x1a, 0x1a, 0x1a, 0x1a, 0x14, 0x15, 0x1a, 0x1a, /* X'38' - X'3F' */
         0x20, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'40' - X'47' */
         0x1a, 0x1a, 0x60, 0x2e, 0x3c, 0x28, 0x2b, 0xe4, /* X'48' - X'4F' */
         0x26, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'50' - X'57' */
         0x1a, 0x1a, 0x21, 0x24, 0x2a, 0x29, 0x3b, 0x1a, /* X'58' - X'5F' */
         0x2d, 0x2f, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'60' - X'67' */
         0x1a, 0x1a, 0x5e, 0x2c, 0x25, 0x5f, 0x3e, 0x3f, /* X'68' - X'6F' */
         0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'70' - X'77' */
         0x1a, 0x1a, 0x3a, 0x23, 0x40, 0x27, 0x3d, 0x22, /* X'78' - X'7F' */
         0x1a, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, /* X'80' - X'87' */
         0x68, 0x69, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'88' - X'8F' */
         0x1a, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, /* X'90' - X'97' */
         0x71, 0x72, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x20ac,/* X'98' - X'9F'*/
         0x1a, 0x1a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, /* X'A0' - X'A7' */
         0x79, 0x7a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'A8' - X'AF' */
         0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'B0' - X'B7' */
         0x1a, 0x1a, 0x1a, 0xd6, 0xc4, 0xdc, 0x1a, 0x1a, /* X'B8' - X'BF' */
         0x1a, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, /* X'C0' - X'C7' */
         0x48, 0x49, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'C8' - X'CF' */
         0x1a, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50, /* X'D0' - X'D7' */
         0x51, 0x52, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'D8' - X'DF' */
         0x1a, 0x1a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, /* X'E0' - X'E7' */
         0x59, 0x5a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, 0x1a, /* X'E8' - X'EF' */
         0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, /* X'F0' - X'F7' */
         0x38, 0x39, 0x1a, 0xf6, 0x1a, 0xfc, 0x1a, 0xdf  /* X'F8' - X'FF' */
         };
```

In Cobol the same table looks like the following:

```
01 EBCDIC-TO-UTF8-TAB.
  03 TAB-00-07 PIC N(8)  VALUE NX"0000000100020003001A0009001A007F".
  03 TAB-08-0F PIC N(8)  VALUE NX"001A001A001A000B000C000D000E000F".
…
  03 TAB-F0-F7 PIC N(8)  VALUE NX"0030003100320033003400350036003 7".
  03 TAB-F8-FF PIC N(8)  VALUE NX"00380039001A00F6001A00FC001A00DF".
```

10.2.2 Code table UTF-8 -> UserEncoding

The following example of the EDF03DRV table shows the structure of the code table transforming UTF-8 to user encoding. As UTF-8 characters might consist of one up to three bytes, the table is built of several parts and takes into account the default values of the leading bits:

0xxx xxxx	1 byte character
110x xxxx	1st byte of a 2 byte character
1110 xxxx	1st byte of a 3 byte character
10xx xxxx	following byte of a 2 or 3 byte character

For every 1 byte character in UTF-8 with binary value i the corresponding hexadecimale EDF03DRV code is given at position i (0 .. 127) in the table (table part 1).

For the 2 byte characters the conversion is done in two steps:

In the first step depending on the first byte which has a value between C0 and DF the index of a table in table part 4 is read from table part 2. In this 64 byte long table the corresponding hexadecimal value of the character in EDF03DRV is found by means of the 2nd byte value (without the first 2 bits).

For 3 byte characters the conversion is done in three steps:

In the first step depending on the first byte which has a value between E0 und EF the index of a table in table part 4 is read in table part 3. This 64 byte long table accessed with the 2nd byte value (without the first 2 bits) contains the index of a table in table part 4. In this 64 byte long table part the corresponding hexadecimal value of the character in EDF03DRV is found by means of the 3rd byte value (without the first 2 bits).

Characters which are not defined are converted to \x00, as are the indices which are not defined in table part 2 and part 3 and in the tables of the 2nd step during conversion of 3 byte characters. The first of the 64 byte tables in table part 4 only consists of \x00 values, that means, that all characters which are not defined are converted to \x00. All other tables of table part 4 can be build user specific and must be referenced by the corresponding positions in table part 2 and part 3.

```
unsigned char const xmltranscodetable_EDF03DRV[256 + 48 + 4 * 64] = {

/* table part 1: 1 Byte code */
         "\x00\x01\x02\x03\x37\x2d\x2e\x2f\x16\x05\x15\x0b\x0c\x0d\x0e\x0f" /* X'00' - X'0F' */
         "\x10\x11\x12\x13\x3c\x3d\x32\x26\x18\x19\x3f\x27\x1c\x1d\x1e\x1f" /* X'10' - X'1F' */
         "\x40\x5a\x7f\x7b\x5b\x6c\x50\x7d\x4d\x5d\x5c\x4e\x6b\x60\x4b\x61" /* X'20' - X'2F' */
         "\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\x7a\x5e\x4c\x7e\x6e\x6f" /* X'30' - X'3F' */
         "\x7c\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xd1\xd2\xd3\xd4\xd5\xd6" /* X'40' - X'4F' */
         "\xd7\xd8\xd9\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xbb\xbc\xbd\x6a\x6d" /* X'50' - X'5F' */
         "\x4a\x81\x82\x83\x84\x85\x86\x87\x88\x89\x91\x92\x93\x94\x95\x96" /* X'60' - X'6F' */
         "\x97\x98\x99\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xfb\x4f\xfd\xff\x07" /* X'70' - X'7F' */
/* table part2: 2 Byte code 1st step */
    "\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" /* 2 Byte code 1st step */
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
/* table part3: 3 Byte code 1st step */
    "\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" /* 3 Byte code 1st step */
/* table part4: 2/3 Byte code 2nd step: */
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" /* first Byte not valid */
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"

    "\x00\x00\x00\x00\xbc\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" /* first Byte  = C3 */
                                                                       /* X'c380' - X'c38f' */
    "\x00\x00\x00\x00\x00\x00\xbb\x00\x00\x00\x00\x00\xbd\x00\x00\xff" /* X'c390' - X'c39f' */
    "\x00\x00\x00\x00\x4f\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" /* X'c3a0' - X'c3af' */
    "\x00\x00\x00\x00\x00\x00\xfb\x00\x00\x00\x00\x00\xfd\x00\x00\x00" /* X'c3b0' - X'c3bf' */

    "\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" /* 3 Byte code 2nd step */
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"

    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" /* X'E28280' - */
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x9f\x00\x00\x00"
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" /* - X'E282bf' */
};
```

In Cobol the same table looks like the following

```
 01 UTF8-TO-EBCDIC-TAB.
   03 TAB-00-0F PIC N(8) VALUE NX"00010203372D2E2F1605150B0C0D0E0F".
   03 TAB-10-1F PIC N(8) VALUE NX"101112133C3D322618193F271C1D1E1F".

  …
   03 TAB-E2828X PIC N(8) VALUE NX"00000000000000000000000000000000".
   03 TAB-E2829X PIC N(8) VALUE NX"00000000000000000000000000000000".
   03 TAB-E282AX PIC N(8) VALUE NX"000000000000000000000009F000000".
   03 TAB-E282BX PIC N(8) VALUE NX"00000000000000000000000000000000".
```

10.2.3 Declaration of user specific encoding functions

When a code table is built as described above the encoding functions can be implemented with means of UTM-XML .

In C the functions looks like the following:

```
int EDF03DRVToUTF8 (unsigned char* out, int *outlen,
                    const unsigned char* in, int *inlen)
{
    int result;
    result = EBCDxToUTF8 (out, outlen, in, inlen,
                          xmlunicodetable_EDF03DRV);
    return result;
}

int UTF8ToEDF03DRV (unsigned char* out, int *outlen,
                    const unsigned char* in, int *inlen)
{
    int result;
    result = UTF8ToEBCDx (out, outlen, in, inlen,
                          xmltranscodetable_EDF03DRV);
    return result;
}
```

In Cobol the functions looks like the following:

```
      *  encoding functions UTF-8 -> EBCDIC, EBCDIC -> UTF-8
      *
       ENTRY "CVUTOE"   USING OUTBUF OUTLEN INBUF INLEN.
      *
       CALL "UTF8ToEBCDx"
            USING OUTBUF, OUTLEN, INBUF, INLEN, UTF8-TO-EBCDIC-TAB.
       EXIT PROGRAM.
      *
       ENTRY "CVETOU"   USING OUTBUF OUTLEN INBUF INLEN.
      *
       CALL "EBCDxToUTF8"
            USING OUTBUF, OUTLEN, INBUF, INLEN, EBCDIC-TO-UTF8-TAB.
       EXIT PROGRAM.
```

10.3 Handling of XML documents without encoding attribute

In some cases, when processing documents on different operationg systems, it causes problems for a document to contain an encoding attribute. Normally, during the transfer of documents (e.g. with ftp) a code conversion is performed. After that the current character set doesn't correspond to the given encoding attribute. This causes problems for applications processing these documents. On the other side the UTM-XML interface should be able to process documents without encoding attribute encoded in a character set different from UTF-8.

Example:

A document is created in BS2000, e.g. with encoding="EBCDIC", and is transferred to Windows with ftp. Normally textual documents are converted automatically from EBCDIC to an ASCII encoding. Tools such as xml editors can't process these documents because of the attribute encoding="EBCDIC".

To solve this problem the UTM-XML interface should be able to create a document with a specified encoding but without encoding attribute (particularly in BS2000 with a special EBCDIC variant).

This behaviour is in contrast to the processing of the libxml2 parser (see http://xmlsoft.org/encoding.html), who interpretes each document without encoding attribute as UTF-8 encoded.

To get the required behaviour, the encoding "NONE" is introduced with a special treatment.

The following treatment is connected to the alias name "NONE" :

- Before processing an XML document without encoding attribute you must set encoding="NONE" as alias name for the desired encoding with KXLSetEncodingAlias (e.g. for the home encoding EDF04IRV).
- For creating an XML document with a special character set (e.g. EDF04IRV) without encoding attribute, you have to set the encoding attribute to NONE (KXLSetDocEncoding). When converting an XML object to a document (KXLConvObjToDoc) the XML document is created with the character set desired but without encoding attribute.
- For reading a document without encoding attribute (caution: encoding should correspond to the NONE encoding), the UTM-XML interface uses the NONE encoding for conversion to UTF-8 and set the encoding to NONE internally in the XML object. If encoding NONE is not defined, the default encoding is UTF-8. No code conversion is performed.

Example:

```
xmlNodePtr  pXMLroot;
char *      pBuffer;
short       rc =0;

/* Declaration of the alias name NONE */
KXLSetEncodingAlias ( "NONE", "EDF03DRV", &rc);

/* Creation of the document */
pXMLroot      = KXLCreateNewObj("mydoc", {"\0", "\0"}, &rc);
/* Processing of the document */
…
/* Setting of the document encoding NONE */
KXLSetDocEnc ( pXMLroot, "NONE", &rc);

/* Creating the document without encoding attribute */
pBuffer = KXLConvObjToDoc(pXMLroot , NULL, NULL, &rc);
```

Output of the document (in encoding EDF03DRV):

```
<?xml version="1.0"?> <mydoc>…</mydoc>
```

10.4 Processing external documents

It is possible, with certain restrictions, to process documents that were not created withthe UTMXML API (referred to below as 'external') using the function calls described here. The behavioral characteristics are described below. The description lays no claim to completeness, however.

10.4.1 Root node

Documents created with KXLCreateNewObj without specification of name and content have an API-specific root node with tag name UTM-XML (<UTM-XML version="03.0"></UTM-XML>). The root node itself can be read with KXLReadNode. Apart from this, the same restrictions as when accessing other nodes apply.

10.4.2 Structure of the nodes

The object nodes created with this API are structured as follows:

- The specified name of the element becomes the tag name
- The element type (string, to which t_value.pType points) is stored in an attribute with the name 'type'. If this is the empty character string, no type attribute is generated. Internally, elements like this are handled as type=struct (interior nodes) or type=string (leaf nodes).
- The value of the element (string, to which t_value.pValue points) is stored as the value of the node.
- Elements representing array elements (child nodes of arrays with type = array) are stored with the additional attribute 'arrayelem = "y"'.

Example:

```
t_value = {"char","y"};
KXLWrite (pNode, "paid", tval, NULL);
```

creates the node:

```
  ELEMENT paid
      ATTRIBUTE type
           TEXT
                content = char
      TEXT
           content = y
```

and corresponds to the element:

```
  <paid type="char"> y </paid>
```

10.4.3 Writing nodes

1. The type conversion functions described in the API can be replaced freely by user-own conversion functions if, for example, higher float precision is desired.
2. If the user is working without the type conversion functions provided by the present API, it is also possible to write nodes of any type. The user himself must then provide the write call with the desired data in a t_value structure. It is important to make sure that multilevel names (containing '/' or '[..]') do not contain any name parts, except for the last, with external type designations.

   Example with '/':
   In order to write a node 'one/two/three' with value '123' and 'one' with type 'myType1', 'two' with type 'struct', 'three' with type 'myType3', the node must be written explicitly with the user-own type:

   ```
   KXLWrite    pname->'one',          tval.pType->'myType1',   tval.pValue->' '
   KXLWrite    pname->'two/three',    tval.pType->'myType3',   tval-pValue->'123'
   ```

   Example with '[ ]':

Writing a node 'Addr[Smith]' with 'Addr' of type 'Address' and 'Smith' of type 'Person' with value ' Broadwater Road':

KXLWrite	pname->'Addr',	tval.pType->'Address',	tval.pValue->' '
KXLWrite	pname->'[Meier]'	tval.pType->'Person',	tval.pValue->'99 Broadwater Road'

Warning: When writing array elements (such as 'Meier' in the above example), it is important to make sure that a '_' is prefixed internally to the element name (this makes numeric specifications possible).

3. Array elements are arranged in ascending order by (numeric) index. The index of an array element can also be non-numeric, however. In this case the elements are arranged numerically by prefixed digits; indices with the same "number prefix" are sorted lexicographically. Thus, for example, the elements [0],[1],[2],[12],[a],[0a],[01a],[1a],[1b],[1a2],[12a] are stored in the following sequence: [a],[0],[0a],[1],[01a],[1a],[1a2],[1b],[2],[12],[12a]

10.4.4 Reading nodes

As array elements have a byte prefixed internally to the array name, a search is made internally for node names without the 1st byte when names enclosed in square brackets are read.

Example: A node with internal name 'pVALUE' can be read with KXLRead pname->'[VALUE]'.

10.5 Example 1

A complete example showing the object structure is presented in this section.

10.5.1 Program code:

The following C data structure is to be stored in an XML document:

```
typedef struct {      Sdate         date;
               Saddress      address;
               Sarticle      article[3];
               double        total;
               char          paid;
                }Saccount;
```

where

```
typedef struct {      short  day;
               int    month;
                      long   year;
                }Sdate;

typedef struct {      char   name[20];
                      char   street[20];
                      int    postalCode;
                      char   town[20];
               char   phone[20];
               }Saddress;
typedef struct {      char orderNumber[10];
                      char name[20];
               float price;
               int quantity;
               }Sarticle;
```

The program code is as follows:

```
xmlNodePtr   pXMLroot, pXMLact, pXMLstruct, pXMLarr;
char *        pBuffer;
short  rc   =0;
short    day=1;
int      month     =1;
long     year      =2013;
char     name[ ]      = "Max Meier";
char     street[ ]    = "Mondstr. 55";
int      postalCode  = 80808;
```

```
   char     town [ ]    = "München";
   char     phone[ ]    = "089/12345678";
   double   total = 0;
   t_value  tval1 = {NULL, NULL};
   t_value  tval2 = {"\0", "\0"};
   int      i;
   char    Stylesheet[ ] = "href=\"mystyle.ccs\" title=\"Compact\"\
 type=\"text/ccs\"";
   char *   pNewBuffer;

   Sarticle art[] = {
   {"A12345", "eraser", 1.50, 1},
   {"B23456", "pencil", 1.00, 4},
   {"C34567", "CD", 4.50, 1},
    };
   char *field_nr[3] = {"[0]", "[1]", "[2]" };
   char *pName;

   pXMLroot = KXLCreateNewObj("mydoc", tval2, &rc);                        (1)
    /* result pXMLroot is pointer to "mydoc" node */

   /* write node account implicitly, node date explicitly */
   pXMLstruct = KXLWrite(pXMLroot, "account/date", KXLFromStruct(NULL), &rc);  (2)
   /* result pXMLstruct is pointer to struct date node */

   /* write structure Sdate */
   pXMLact=KXLWrite(pXMLroot,"account/date/day",KXLFromShort(day),&rc);
(3)
   pXMLact=KXLWrite(pXMLstruct, "month", KXLFromInt(month),&rc);
  (4)
   pXMLact=KXLWrite(pXMLstruct, "year", KXLFromLong(year),&rc);             (5)
   /* node account exists; create new node address */
   pXMLstruct=KXLWrite(pXMLroot, "account/address", KXLFromStruct(NULL), &rc);
  (6)
   /* pXMLStruct is pointer struct address node */

   /* write structure address */
   pXMLact     =KXLWrite(pXMLstruct, "name", KXLFromString(name),&rc);
  (7)
   pXMLact     =KXLWrite(pXMLstruct, "street", KXLFromString(street),&rc);
  (8)
   pXMLact     =KXLWrite(pXMLstruct, "postalCode", KXLFromInt(postalCode),&rc);
  (9)
   pXMLact     =KXLWrite(pXMLstruct, "town", KXLFromString(town),&rc);
  (10)
   pXMLact     =KXLWrite(pXMLstruct, "phone", KXLFromString(phone),&rc);
  (11)

   pXMLstruct  = KXLSetParentNode(pXMLstruct, &rc);
   /* pXMLstruct now pointer to node account */
   pXMLarr     =KXLWrite(pXMLstruct, "article", KXLFromArray( ),&rc);
(12)
   /* pXMLarr pointer to array article node */

   for (i = 0; i<=2; i++) /* write field elements = structures of type Sarticle */
   {
     pXMLstruct=KXLWrite(pXMLarr, field_nr[i], KXLFromStruct("Sarticle"),&rc);(13)
     /* pXMLstruct pointer to struct [0], [1] or [2]*/
     /* write structure node with value Sarticle */
     pXMLact  =KXLWrite(pXMLstruct, "orderNumber",
                     KXLFromString(art[i].orderNumber),&rc);
                (14)
     pXMLact  =KXLWrite(pXMLstruct, "name", KXLFromString(art[i].name),&rc);  (15)
     pXMLact  =KXLWrite(pXMLstruct, "price", KXLFromFloat(art[i].price),&rc); (16)
     pXMLact  =KXLWrite(pXMLstruct,"quantity",
                          KXLFromInt(art[i].quantity),&rc);                   (17)
     /* end of write structure elements */
     /* calculate accumulated total */
     total = total + art[i].price * art[i ].quantity;
   }
   /* write total and paid nodes */
   pXMLact =KXLWrite( pXMLroot, "account/total", KXLFromDouble(total),&rc);   (18)
   pXMLact =KXLWrite( pXMLroot, "account/paid", KXLFromChar('n'),&rc);        (19)
   pBuffer = KXLConvObjToDoc(pXMLroot ,(char*)&Stylesheet,NULL, &rc);         (20)
   /* pBuffer  is pointer to the buffer containing the converted object */
   /* now you may send pBuffer with XML-document e.g. to a communication partner */

   /* examples for read calls: */
   /* read the root node */
   tval1 = KXLReadNode(pXMLroot, KXL_FALSE, &pName, &rc);                     (21)
   /* 2 possible read calls, e.g. the name of the first article: */
   /* read directly */
   tval1 = KXLRead (pXMLroot, "account/article[0]/name",KXL_FALSE, &rc);      (22)
   /* or position and read */
   pXMLstruct = KXLSetSubObject (pXMLroot, "account/article[0]", &rc);        (23)
   tval2 = KXLRead (pXMLstruct, "name", KXL_FALSE, &rc);                      (24)

   /* end handling: */
   pXMLact = pXMLstruct = pXMLarr = NULL;  /* clear pointer */
   KXLFreeObj(pXMLroot, &rc);                     /* free XML-objects */       (25)
   pXMLroot = NULL;
```

10.5.2 Program trace

Extracts from a trace file generated with Tracemode=F for the above example appears as shown below. The numbers in parentheses at the right-hand edge refer to the calls in the program, while the letters in parentheses refer to notes following the trace extracts.

```
/XML for UTM V03.0A0  -- TRACE 1 -- PID:02356 -- Wed Nov 23 13:23:23 2005 (a)
WIN32 osversion=2600 winversion=5.1 cpumode=protected mode
Tracemode: Full, Maximum Size: 16384 KByte

13:23:23 KXLTSENV
         Set Trace environment:
13:23:23 KXLTSENV: Trace initiated
13:23:23 KXLCreateNewObj init: N=mydoc, T=, V=                            (1)
13:23:23 KXLInitEnv init
13:23:23 KXLCheckInitVersion init, Version 30
13:23:23 KXLCheckInitVersion exit,Parser V=20620, RC=0
13:23:23 KXLCheckCreaVersion init, Version 30
13:23:23 KXLCheckCreaVersion exit,Parser V=20620, RC=0
13:23:23 KXLCheckConvVersion init, Version 30
13:23:23 KXLCheckConvVersion exit, RC=0
13:23:23 KXLCheckReadVersion init, Version 30
13:23:23 KXLCheckReadVersion exit, RC=0
13:23:23 KXLCheckTracVersion init, Version 30
13:23:23 KXLCheckTracVersion exit, RC=0
13:23:23 KXLCheckScmaVersion init, Version 30
13:23:23 KXLCheckScmaVersion exit, RC=0
13:23:23 KXLInitEncHdlr init
13:23:23 KXLInitEncHdlr exit, RC=0
13:23:23 KXLInitEnv exit,
13:23:23 KXLAnalyseName init, N=mydoc
13:23:23 KXLAnalyseName exit, N=, LTH=5, ET=1, P_LTH=-1
...
13:23:23 KXLCreateNewObj exit,XML-A=00325F28, RC=0
13:23:23 KXLFromStruct : init, ST=(null)
13:23:23 KXLFromStruct : exit, T=struct, V=(null)
13:23:23 KXLWrite init, XML-A=00325F28, N=account/date, T=struct, V=      (2)
...
13:23:23 KXLWrite exit,XML-A=003263E0,RC=0
13:23:23 KXLFromShort : init, SH=   1
13:23:23 KXLFromShort : exit, T=short, V=1
13:23:23 KXLWrite init, XML-A=00325F28, N=account/date/day, T=short, V=1  (3)
...
13:23:23 KXLWrite exit,XML-A=003268F8,RC=0
13:23:23 KXLFromInt : init, IN=       1
13:23:23 KXLFromInt : exit, T=int, V=1
13:23:23 KXLWrite init, XML-A=003263E0, N=month, T=int, V=1               (4)
...
13:23:23 KXLWrite exit,XML-A=00326B80,RC=0
13:23:23 KXLFromLong : init, LG=     7D6
```

```
13:23:23 KXLFromLong : exit, T=long, V=2006
13:23:23 KXLWrite init, XML-A=003263E0, N=year, T=long, V=2006            (5)
...
13:23:23 KXLWrite exit,XML-A=00326EE8,RC=0
13:23:23 KXLFromStruct : init, ST=(null)
13:23:23 KXLFromStruct : exit, T=struct, V=(null)
13:23:23 KXLWrite init, XML-A=00325F28, N=account/address, T=struct, V=   (6)
...
13:23:23 KXLWrite exit,XML-A=00327308,RC=0
13:23:23 KXLFromString : init, ST=Max Meier
13:23:23 KXLFromString : exit, T=string, V=Max Meier
13:23:23 KXLWrite init, XML-A=00327308, N=name, T=string, V=Max Meier     (7)
...
13:23:23 KXLWrite exit,XML-A=003277C8,RC=0
13:23:23 KXLFromString : init, ST=Mondstr. 55
13:23:23 KXLFromString : exit, T=string, V=Mondstr. 55
13:23:23 KXLWrite init, XML-A=00327308, N=street, T=string, V=Mondstr. 55 (8)
...
13:23:23 KXLWrite exit,XML-A=00327F20,RC=0
13:23:23 KXLFromInt : init, IN=   13BA8
13:23:23 KXLFromInt : exit, T=int, V=80808
13:23:23 KXLWrite init, XML-A=00327308, N=postalCode, T=int, V=80808      (9)
...
13:23:23 KXLWrite exit,XML-A=00327CA0,RC=0
13:23:23 KXLFromString : init, ST=München
13:23:23 KXLFromString : exit, T=string, V=München
13:23:23 KXLWrite init, XML-A=00327308, N=town, T=string, V=München      (10)
...
13:23:23 KXLWrite exit,XML-A=00328130,RC=0
13:23:23 KXLFromString : init, ST=089/12345678
13:23:23 KXLFromString : exit, T=string, V=089/12345678
13:23:23 KXLWrite init, XML-A=00327308, N=phone, T=string, V=089/12345678 (11)
...
13:23:23 KXLWrite exit,XML-A=003284A8,RC=0
13:23:23 KXLSetParentNode init, XML-A=00327308
13:23:23 KXLSetParentNode exit, XML-A=00326228, RC=0
13:23:23 KXLFromArray : init
13:23:23 KXLFromArray : exit, T=array
13:23:23 KXLWrite init, XML-A=00326228, N=article, T=array, V=           (12)
...
13:23:23 KXLWrite exit,XML-A=00328D70,RC=0
13:23:23 KXLFromStruct : init, ST=Sarticle
13:23:23 KXLFromStruct : exit, T=struct, V=Sarticle
13:23:23 KXLWrite init, XML-A=00328D70, N=[0], T=struct, V=Sarticle      (13)
..
13:23:23 KXLWrite exit,XML-A=00328890,RC=0
13:23:23 KXLFromString : init, ST=A12345
13:23:23 KXLFromString : exit, T=string, V=A12345
13:23:23 KXLWrite init, XML-A=00328890, N=orderNumber, T=string, V=A12345 (14)
...
13:23:23 KXLWrite exit,XML-A=003291C0,RC=0
13:23:23 KXLFromString : init, ST=eraser
13:23:23 KXLFromString : exit, T=string, V=eraser
13:23:23 KXLWrite init, XML-A=00328890, N=name, T=string, V=eraser       (15)
...
13:23:23 KXLWrite exit,XML-A=00329568,RC=0
13:23:23 KXLFromFloat : init, FL=        0000c03f
|...?            |
13:23:23 KXLFromFloat : exit, T=float, V=1.5
13:23:23 KXLWrite init, XML-A=00328890, N=price, T=float, V=1.5          (16)
...
13:23:23 KXLWrite exit,XML-A=003298E0,RC=0
13:23:23 KXLFromInt : init, IN=       1
13:23:23 KXLFromInt : exit, T=int, V=1
13:23:23 KXLWrite init, XML-A=00328890, N=quantity, T=int, V=1           (17)
...
13:23:23 KXLWrite exit,XML-A=00329C00,RC=0
...
13:23:23 KXLFromDouble : init, DO=         00000000 00002440
|......$@        |
13:23:23 KXLFromDouble : exit, T=double, V=10.
13:23:23 KXLWrite init, XML-A=00325F28, N=account/total, T=double, V=10. (18)
...
```

```
13:23:23 KXLWrite exit,XML-A=0032C830,RC=0
13:23:23 KXLFromChar : init, CH=n
13:23:23 KXLFromChar : exit, T=char, V=n
13:23:23 KXLWrite init, XML-A=00325F28, N=account/paid, T=char, V=n      (19)
...
13:23:23 KXLWrite exit,XML-A=0032CA68,RC=0
13:23:23 KXLConvObjToDoc init, XML-A=00325F28, S=href="mystyle.ccs" title="Compact"
type="text/ccs"  (20)
...
13:23:23 Output of the converted XML document:                            (b)
<?xml version="1.0" encoding="MSDOSLat"?>
<?xml-stylesheet href="mystyle.ccs" title="Compact" type="text/ccs"?>
<mydoc>
  <account type="struct">
    <date type="struct">
      <day type="short">1</day>
      <month type="int">1</month>
      <year type="long">2006</year>
    </date>
    <address type="struct">
      <name type="string">Max Meier</name>
      <street type="string">Mondstr. 55</street>
      <postalCode type="int">80808</postalCode>
      <town type="string">München</town>
      <phone type="string">089/12345678</phone>
    </address>
    <article type="array">
      <_0 type="struct" ArrayElem="y">Sarticle
        <orderNumber type="string">A12345</orderNumber>
        <name type="string">eraser</name>
        <price type="float">1.5</price>
        <quantity type="int">1</quantity>
      </_0>
      <_1 type="struct" ArrayElem="y">Sarticle
        <orderNumber type="string">B23456</orderNumber>
        <name type="string">pencil</name>
        <price type="float">1.</price>
        <quantity type="int">4</quantity>
      </_1>
      <_2 type="struct" ArrayElem="y">Sarticle
        <orderNumber type="string">C34567</orderNumber>
        <name type="string">CD</name>
        <price type="float">4.5</price>
        <quantity type="int">1</quantity>
      </_2>
    </article>
    <total type="double">10.</total>
    <paid type="char">n</paid>
  </account>
</mydoc>

13:23:23 KXLConvObjToDoc exit, DOC-A=0032F280, LTH=1180, RC=0      ende (20)
13:23:23 KXLReadNode init, XML-A=00325F28, FULL=0                       (21)
...
13:23:23 KXLReadNode exit, N=mydoc, T=, V=, RC=0
13:23:23 KXLRead init, XML-A=00325F28, N=account/article[0]/name, DEL=0 (22)
...
13:23:23 KXLRead exit, T=string, V=eraser, RC=0
13:23:23 KXLSetSubObject init, XML-A=00325F28, N=account/article[0]      (23)
...
13:23:23 KXLSetSubObject exit, XML-A=00328890, RC=0
13:23:23 KXLRead init, XML-A=00328890, N=name, DEL=0                     (24)
...
13:23:23 KXLRead exit, T=string, V=eraser, RC=0
13:23:23 KXLFreeObj init, XML-A=00325F28                                 (25)
13:23:23 KXLFreeObj exit, RC=0
```

Notes:

(a) Header of the trace file

(b) If Tracemode=F is specified, the converted XML document is also logged; the document is formatted here so it can be read more easily.

10.6 Example 2

This section contains a further example, with some new functions of V2.0 (namespaces, attributes)

In this example, the following document is processed.

```
<?xml version="1.0" encoding="MSDOSLat"?>
<account xmlns="http://www.mydoc.de"\
          xmlns:adr="http://www.myaddr.de"
                customerNumber="12345678" status="paid">
  <date>20/03/2013</date>
  <adr:address type="struct">
    <adr:name >Max Meier</adr:name>
    <adr:street >Mondstr. 55</adr:street>
    <adr:postalCode >80808</adr:postalCode>
    <adr:town >München</adr:town>
    <adr:phone>089/12345678</adr:phone>
    </adr:address>
  <article section="office">
    <orderNumber>A12345</orderNumber>
    <name>eraser</name>
    <price currency="Euro">1.5</price>
    <quantity>1</quantity>
  </article>
  <article  section="office">
    <orderNumber>B23456</orderNumber>
    <name>pencil</name>
    <price currency="Euro">1.</price>
    <quantity>4</quantity>
  </article>
  <article  section="music">
    <orderNumber>C34567</orderNumber>
    <name>CD</name>
    <price currency="Euro">4.5</price>
    <quantity>1</quantity>
  </article>
  <deliveryCharges currency="Euro">2.</deliveryCharges>
  <total currency="Euro">12.</total>
</account>
```

10.6.1 Program code

The following subfunctions are used:

```
void stock_office(char * orderNumber, int quantity)
void stock_music(char * orderNumber, int quantity)
void stock_house(char * orderNumber, int quantity)
void finances(char * customer_no, float deliveryCharge, double total)
```

The program code is as follows:

```
float conv_to_euro (char * val, float f)
static void  print_RC_and_msg (char * apiCall, short rc);

xmlNodePtr    pXMLroot, pXMLact, pXMLattr;
short rc=0;
char *pName;
char customerNumber[10]="\0";
char orderNumber[10]="\0";
int status = 0;
int i;
float f, deliveryCharges = 0;
double total=0;
t_value tval;
char *pPrintBuffer;

pXMLroot    =KXLConvDocToObj(pBuffer, &rc);                          (a)
/* check returncode */
if (rc != KXL_RC_OK)
{ print_RC_and_msg ("KXLConvDocToObj", rc);
  return;
}
```

```
/* read first attribute of the root node */
tval = KXLReadAttr(pXMLroot, &pName, &pXMLact, &rc);                (b)
while (rc == KXL_RC_OK)                                             (b)
{ /* read custumer number and status */
    if (strcmp (pName, "@customerNumber")== 0)
         strcpy (customerNumber, tval.pValue);
    if (strcmp (pName, "@status")== 0)
         if (strcmp (tval.pValue,"paid")==0)
             status = 1;
    /* read next attribute */
    tval = KXLReadNextSib(pXMLact, &pName, &pXMLact, &rc);          (b)
}

if ( status)
/* process all articles in loop, if account is paid */
{ /* set to first article */
    pXMLact = KXLSetSubObject (pXMLroot, "article", &rc);           (c)
    while (rc == KXL_RC_OK)
        { /* read parameter and pass them on to processing */
        tval = KXLRead (pXMLact, "orderNumber", KXL_FALSE, &rc);
        strcpy(orderNumber, tval.pValue);
        tval = KXLRead (pXMLact, "quantity", KXL_FALSE, &rc);
        i = KXLToInt(tval, NULL);
        tval = KXLRead (pXMLact, "@section", KXL_FALSE, &rc);
        if (strcmp(tval.pValue, "office")==0)
            stock_office (orderNumber, i);

        if (strcmp(tval.pValue, "music")==0)
            stock_music (orderNumber, i);

        if (strcmp(tval.pValue, "house")==0)
            stock_house (orderNumber, i);

        /* set to next article ( = read all elements of the nodelist )*/
        pXMLact = KXLSetSubObject (pXMLact, "[+]", &rc);            (c)
           }

    /* read deliveryCharges and total */
          tval = KXLRead (pXMLroot, "deliveryCharges", KXL_FALSE, &rc);
    f = KXLToFloat (tval, NULL);
    tval = KXLRead (pXMLroot, "deliveryCharges/@currency",
                  KXL_FALSE, &rc);                                   (d)
    if (strcmp (tval.pValue, "Euro")==0)
        deliveryCharges = f;
    else
        deliveryCharges = conv_to_euro (tval.pValue, f);

    tval = KXLRead (pXMLroot, "total", KXL_FALSE, &rc);
    f = KXLToFloat (tval, NULL);
    tval = KXLRead (pXMLroot, "total/@currency", KXL_FALSE, &rc);    (d)
    if (strcmp (tval.pValue, "Euro")==0)
        total = f;
    else
        total = conv_to_euro (tval.pValue, f);

    /* passing on to financial section */
    finances (customerNumber, deliveryCharges, total);
}
/* clear pointer */
pXMLact =  pXMLattr = NULL;
/* free XML object */
KXLFreeObj(pXMLroot, &rc);
pXMLroot = NULL;
```

Comments on the program code:

(a) Document is converted.
(b) All attributes of the node are read: the first is read with KXLReadAttr, all others with KXLReadNextSib, where the returned element addresses are used for the next call. The namespace definitions are not read here.
(c) All elements of a node list are processed: the pointer is positioned on the first element when the name is specified; '[+]' is used as a name for further position calls.
(d) Attributes are read directly.

10.7 Example 3

This section contains an example using the schema functions.

The following document can be found in pXMLDocBuffer for this purpose:

```
<?xml version="1.0" encoding="UTF-8"?>
<adr:addressbook xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:adr = "http://www.xml.test/test/ADR"
                      xsi:schemaLocation="http://www.xml.test/test/ADR
                                    http://www.xml.test/test/adr.xsd">
  <adr:address>
    <adr:name>Anke Alpha</adr:name>
    <adr:street>Astreet</adr:street>
    <adr:postalCode>A1234</adr:postalCode>
    <adr:town>Atown</adr:town>
    <adr:email>Alpha@web.de</adr:email>
  </adr:address>
  <adr:address>
    <adr:name>Bodo Beta</adr:name>
    <adr:street>Bstreet</adr:street>
    <adr:postalCode>B5678</adr:postalCode>
    <adr:town>Btown</adr:town>
  </adr:address>
</adr:addressbook>
```

as well as the following schema in pSchemaBuffer:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.xml.test/test/ADR"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:adr="http://www.xml.test/test/ADR"
        xmlns="http://www.w3.org/2001/XMLSchema"
              version = "1.0.0" elementFormDefault = "qualified">
<xsd:element name="person" type="adr:addressTypeDe"/>
<xsd:element name="addressbook">
 <xsd:complexType>
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
   <xsd:element name="address" type="adr:addressTypeDe"
                minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
<xsd:complexType name="addressTypeDe">
 <xsd:sequence minOccurs="1" maxOccurs="1">
  <xsd:element name="name" type="xsd:string" minOccurs="1" maxOccurs="1"/>
  <xsd:element name="street" type="xsd:string" minOccurs="1" maxOccurs="1"/>
  <xsd:element name="postalCode" type="xsd:string"
               minOccurs="1" maxOccurs="1"/>
  <xsd:element name="town" type="xsd:string" minOccurs="1" maxOccurs="1"/>
  <xsd:element name="phone" type="xsd:string"
               minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="email" type="xsd:string"
               minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
</schema>";
```

10.7.1 Program code

The program code appears as follows, where

- create_printable_doc puts the document in a printable format
  `char *  create_printable_doc (char * pBuffer);`
  and
- print_RC_and_msg output the return code and any eventual parser messages.
  `void  print_RC_and_msg (char * apiCall, short rc);`

```
xmlNodePtr    pXMLDoc;
xmlSchemaPtr  pXMLSchema;
char*         pXmlDocBuffer;
char*         pSchemaBuffer;
short rc=0;
char *pLocalDir = "";
char *pPrintBuffer;

 pPrintBuffer = create_printable_doc(pXMLDocBuffer);
 printf ("xml document is:\n %s\n", pPrintBuffer);
 pPrintBuffer = create_printable_doc(pSchemaBuffer);
 printf ("schema document is:\n %s\n", pPrintBuffer);

 /* convert document to DOM object */
 pXMLDoc = KXLConvDocToObj(pXMLDocBuffer, &rc);
 print_RC_and_msg ("KXLConvDocToObj", rc);
 /* check returncode */
 if (rc != KXL_RC_OK)
   return;

 /* parse schema from buffer */
 pXMLSchema = KXLParseSchema (pSchemaBuffer, pLocalDir, &rc);
 print_RC_and_msg ("KXLParseSchema", rc);
 /* same as following call with file adr.xsd containing the schema
    pXMLSchema = KXLParseSchemaFile ("adr.xsd", pLocalDir, &rc);
    print_RC_and_msg ("KXLParseSchemaFile", rc);
 /* */
 /* check returncode */
 if (rc != KXL_RC_OK)
   return;

 /* validate xml object vs. schema */
 KXLValidDoc (pXMLDoc, pXMLSchema, &rc);
 print_RC_and_msg ("KXLValidDoc", rc);
 /* instead of KXLConvDocToObj and KXLValidDoc also do the following call
    if you don't need pXMLDoc:
    KXLValidDocBuf (pXMLDocBuffer, pXMLSchema, &rc);
    print_RC_and_msg ("KXLValidDocBuf", rc);
 /* */

/* validation of XML document with Schema file adr.xsd in local directory */
 pXMLDoc =KXLConvDocToObjAndValid(pXMLDocBuffer,KXL_TRUE, pLocalDir, &rc);
 print_RC_and_msg ("KXLConvDocToObjAndValid", rc);

 /* free pointer to doc and schema object */
 KXLFreeSchema(pXMLSchema, NULL);
 KXLFreeObj(pXMLDoc, NULL);
```

10.8 Literature references

[DOM specification]	DOM specification published by the W3C (www Consortium) Webpage: http://www.w3.org/DOM
[NS specification]	Name space – Recommendation of the W3C (www – Consortium) web – page: http://www.w3.org/TR/REC-xml-names/
[XML specification]	XML specification of the W3C (www Consortium) Webpage: http://www.w3.org/XML
[XML-Schema – Definition]	http://en.wikipedia.org/wiki/XML_Schema
[XML-Schema – Specification]	web pages: http://www.w3.org/TR/xmlschema-0/ http://www.w3.org/TR/xmlschema-1/ http://www.w3.org/TR/xmlschema-2/
[GNOME - Parser]	Parser GNOME XML –library V1.0, freeware from Daniel Veillard web – page: http://xmlsoft.org
[UTF8]	http://en.wikipedia.org/wiki/UTF-8
[libxml internalization support]	http://xmlsoft.org/encoding.html
[IO-Interfaces]	http://xmlsoft.org/xmlio.html