The XMP Toolkit

Version 2.8

September 14, 2001
Chapter 1  Preface  .............................................................. 1
  1.1 About This Document ................................................. 1
  1.2 Audience ........................................................................ 1
  1.3 Assumptions .................................................................... 1
  1.4 How This Document Is Organized .................................... 1
  1.5 Conventions used in this Document ................................. 2
  1.6 Where to Go for More Information ................................. 2

Chapter 2  The XMP Toolkit .................................................. 3
  2.1 Overview ........................................................................ 3
  2.2 The XMP Toolkit ............................................................ 3
  2.3 Implementation Notes .................................................... 4
      2.3.1 Overview ............................................................... 4
      2.3.2 Construction and Destruction ................................. 4
      2.3.3 Memory Management ............................................. 5
      2.3.4 Style and Conventions ........................................... 5

Chapter 3  MetaXAP .............................................................. 9
  3.1 MetaXAP Overview ...................................................... 9
  3.2 Introduction .................................................................... 9
  3.3 Path Composition .......................................................... 10
      3.3.1 XPath Syntax ....................................................... 12
  3.4 Property Value Features ................................................ 14
  3.5 Standard Attributes ...................................................... 15
  3.6 MetaXAP Class ............................................................. 15
      3.6.1 Storage Management ............................................ 15
  3.7 Important Types Used In MetaXAP ................................. 16
      3.7.1 Namespace Constants ............................................ 18
  3.8 MetaXAP Member Functions .......................................... 19
  3.9 MetaXAP Static Functions (Class Methods) ..................... 37
  3.10 XAPPaths Class ......................................................... 43
## Contents

<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>UtilityXAP</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>UtilityXAP</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>UtilityXAP Static Functions (Class Methods)</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix A</th>
<th>XMP Toolkit Exceptions</th>
<th>55</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Appendix B</th>
<th>Runtime Flow of Control</th>
<th>59</th>
</tr>
</thead>
</table>

| Appendix C | XMP Toolkit Function List | 69 |
1 Preface

1.1 About This Document

This Preface contains information about this document, describes its organization and the conventions used in the document, and where to go for additional information.

1.2 Audience

The audience for this document includes developers of applications who have licensed the XMP Toolkit.

1.3 Assumptions

This document assumes that you are familiar with the XMP specification, and that you are familiar with C++ and an appropriate development environment.

1.4 How This Document Is Organized

In addition to this preface, this document consists of the following chapters:

Chapter 2: The XMP Toolkit
Contains an overview of the XMP Toolkit, and a short section on implementation notes.

Chapter 3: MetaXAP
Describes the MetaXAP Class, which provides tools for reading, writing, and manipulating XMP metadata. MetaXAP is the primary interface to the XMP Toolkit.

Chapter 4: UtilityXAP
Describes the UtilityXAP class, a variety of special purpose utilities to simplify common uses of MetaXAP.

Appendix A: XMP Toolkit Exceptions
Lists the C++ exceptions that can be raised through the use of the XMP Toolkit member functions.

Appendix B: Runtime Flow of Control
Provides a detailed roadmap that follows the most important paths through the code.
Appendix C: XMP Toolkit Function List
Lists the XMP Toolkit functions along with a brief description of what each one does.

1.5 Conventions used in this Document

The following type styles are used for specific types of text:

<table>
<thead>
<tr>
<th>Typeface Style</th>
<th>Used for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serifed Roman Italic Caps</td>
<td>Values. For example, TRUE, NULL, etc.</td>
</tr>
<tr>
<td>Sans serif bold</td>
<td>XMP property names. (Always prefaced with “xap” and a single colon. For example: xap:MetadataDate.</td>
</tr>
<tr>
<td>Monospaced Regular</td>
<td>All C++ Code, function parameters, file names, etc.</td>
</tr>
<tr>
<td>Monospaced Bold</td>
<td>Member function names in text</td>
</tr>
</tbody>
</table>

1.6 Where to Go for More Information

The main reference to be used in conjunction with this document is XMP – Extensible Metadata Platform, which contains the specification of XMP schemas, properties, value types, and the interchange format.

In addition, the following Internet standard may be of use (a longer list of standards used in XMP is included in XMP – Extensible Metadata Platform):

IETF Standard for Language Element Values (RFC 1766):
http://www.ietf.org/rfc/rfc1766.txt?number=1766
2.1 Overview

This document describes the XMP Toolkit which was designed to help applications with handling XMP operations such as the creation and manipulation of metadata. The availability of the Toolkit makes it easier for developers to support XMP metadata, and helps to standardize how the data is represented and interchanged. The XMP Toolkit can be licensed, royalty-free, from Adobe Systems.

This chapter includes a brief overview of the key features of the Toolkit and provides some basic implementation notes.

2.2 The XMP Toolkit

The XMP Toolkit features a C++ interface which uses some modern (ANSI) features, such as exceptions, STL strings, and bool. It uses conservative coding and interface design for maximum portability and to make it easier for applications to adopt.

**NOTE:** Many namespaces, keywords, and related names in this document are prefaced with the string “XAP”, which was an early internal code name for XMP metadata. Because Acrobat 5.0 used those names, they were retained for compatibility purposes.

The XMP Toolkit consists of two parts:

- **MetaXAP** manages the metadata for a managed resource such as an application document file. It defines the objects that act as containers for properties relating to a specific document, and is the primary interface to the XMP Toolkit. MetaXAP provides the top level abstraction for metadata about a document. Nodes are accessed via string pathnames which use a simplified form of XPath strings (XML Path Language: http://www.w3.org/TR/xpath)

- **UtilityXAP** provides a variety of special purpose utilities to simplify common uses of MetaXAP. For example, MetaXAP reads and writes property values as strings. UtilityXAP has services that include conversion to or from integers and other types.

XMP metadata properties are organized by schemas (see *The XMP Metadata Framework* for more information about XMP schemas). In RDF, the schema is defined by a namespace attribute. Within each schema, properties are named via a *path string*. This path string has a very simple syntax which is modelled on the XPath standard.

The full XMP data model is supported, including values that are simple literals, nested descriptions, and structured containers. Applications should include only the *XAPToolkit.h* file, and optionally the *UtilityXAP.h* file.

In addition, the following points apply to the XMP Toolkit:
2.3 Implementation Notes

The following sections give an overview of how the Toolkit is put together. You should read this document in combination with the comments in various header files. Begin with the implementation notes in this chapter, and then progress to the introductory sections of the chapters on “MetaXAP” (sections 3.1 through 3.6), and finally, “UtilityXAP.” For a detailed view of how the XMP Toolkit works, see Appendix B, “Runtime Flow of Control.”

2.3.1 Overview

The XMP Toolkit implements one main object, MetaXAP. UtilityXAP is a collection of static utility functions. Various smaller objects, such as XAPClock and XAPPaths, are used to support the main objects. They are described later in this document.

2.3.2 Construction and Destruction

Most clients start by constructing a MetaXAP object.

As explained in the MetaXAP.h header file, MetaXAP is a Handle class. The only member variable is the opaque XAPTk_Data* m_data. At construction time, a new XAPTk_Data object is created. See XAPTkData.h for its member variables, which are initialized on construction.

The MetaXAP constructor that takes a XAPClock creates an object capable of tracking changes to the metadata with timestamps.

The MetaXAP constructor that also takes a buffer of XML is a convenience. It is equivalent to calling the default constructor, and immediately calling MetaXAP::parse.
A MetaXAP object can be used without parsing. You just create properties in it with `MetaXAP::set` and `MetaXAP::createFirstItem`. However, most objects will be filled up by parsing XML. This is done with the `MetaXAP::parse` function.

Copy construction for MetaXAP is prohibited. Instead, a `Clone` static function is provided. These objects manage large and complex data structures; making unintentional copy construction very expensive, which is why they are prohibited.

Destruction deletes the `XAPTk_Data` object, which in turn deletes all of the memory allocated for its member variables.

### 2.3.3 Memory Management

Any non-const data structure returned to the client is a copy. It is up to the client to free it. `Const` structures are owned by the XMP Toolkit.

When strings and other data structures are output parameters for functions, they are specified as non-const reference variables. This guarantees that storage control remains with the client. Direct return of objects is avoided in order to avoid unintended copy construction.

### 2.3.4 Style and Conventions

The following is an unordered list of items that will help you understand and navigate through the code.

**Naming Styles**

Table 2.1, “Naming Styles used in the XMP Toolkit” lists the naming styles used for the XMP Toolkit.

<table>
<thead>
<tr>
<th>Item</th>
<th>Naming Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>TitleCase, sometimes with: Prefix_Underbar</td>
</tr>
<tr>
<td>Module Functions</td>
<td>TitleCase</td>
</tr>
<tr>
<td>Class Static Functions</td>
<td>TitleCase (ClassName::TitleCase)</td>
</tr>
<tr>
<td>Member Functions</td>
<td>initialLowerMixedCase</td>
</tr>
<tr>
<td>Public Enum Members</td>
<td>xap_all_lower_case_with_underbars</td>
</tr>
<tr>
<td>Private Enum Members</td>
<td>initialLowerMixedCase</td>
</tr>
</tbody>
</table>
Names Of Constants And Types

Public types, particularly enums, begin with “XAP” with no underbar. Examples are XAPFeatures and XAPStructContainerType. Most are defined in XAPDefs.h, though a few are defined in the class header file that they are most closely associated with.

Public constants, such as namespace names, begin with “XAP”. For example, XAP_NS_XAP. These are also defined in XAPDefs.h.

Public enum members begin with “xap_”. For example, xap_bag.

Package constants begin with “XAPTk_”. Most are defined in XAPTkdefs.h. For example: XAPTk_ATTR_XML_LANG.

Names Of Exceptions

With the exception of xap_no_match, all exceptions begin “xap_bad” and are derived from either xap_error (same sense as the Java Error class), or xap_client_fault (same sense as the Java Exception class). See XAPExcept.h.

XAPTk_Composite Types, Module Symbols

The header file XAPObjWrapper.h contains data-structure typedefs built up from STL building blocks. The naming convention for these, are as follows:

### TABLE 2.2 Typedef Naming Convention

<table>
<thead>
<tr>
<th>STL</th>
<th>Name Pattern</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::map</td>
<td>XAPTk_{Foo}By{Bar}</td>
<td>XAPTk_StringByString</td>
</tr>
<tr>
<td>std::vector</td>
<td>XAPTk_VectorOf{Foo}</td>
<td>XAPTk_VectorOfString</td>
</tr>
<tr>
<td>std::pair</td>
<td>XAPTk_PairOf{Foo}</td>
<td>XAPTk_PairOfString</td>
</tr>
<tr>
<td>std::stack</td>
<td>XAPTk_StackOf{Foo}</td>
<td>XAPTk_StackOfString</td>
</tr>
</tbody>
</table>

Where {Foo} and {Bar} are one of the abbreviations (that is, either the “String” or “Pair”) in the second column in the following table:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::string</td>
<td>String</td>
</tr>
<tr>
<td>std::pair</td>
<td>Pair</td>
</tr>
</tbody>
</table>

There are also some types in the XAPTk::namespace that are more implementation specific. In these cases {Foo} or {Bar} describe the intended usage, rather than the base type, for example:
XAPTk::VectorOfProps
XAPTk::StackOfNSDefs

The name of a class is used as a prefix for *package* global functions. For example, 
MetaXAP_CollectAliases is a global function defined in MetaXAP.cpp.

On the other hand, when `{class}_` is used as a prefix for variables, it means they are module 
static (private). For example, MetaXAP_nsMap is a static module function of MetaXAP.cpp.

As described above, XAPTk_ was used prior to the introduction of the XAPTk::namespace.
3.1 MetaXAP Overview

This section describes the MetaXAP Class of the XMP Toolkit, which is used to read, write, and manipulate XMP metadata embedded in, or associated with, managed resources.

3.2 Introduction

MetaXAP is a container class equivalent to the <RDF>... </RDF> element.

A single instance of the MetaXAP class represents the metadata about one resource (application file). A MetaXAP object contains the internal tree representation of a parsed XML stream of XMP metadata. The nodes of this tree are accessed through namespace and pathname strings. Input and output is based on a very simple cross-platform buffer-stream mechanism. Basically, you construct MetaXAP with a buffer of XML, you do read/writes on the in-memory model, and then you get a buffer of potentially modified XML back.

MetaXAP enables clients to:

- define namespaces
- get and set property values and attributes
- parse existing RDF metadata
- serialize a MetaXAP object to RDF
- enumerate all of the properties, associated with a resource, by schema, or all properties at and below a specified partial path

MetaXAP also provides a static set of known schema namespace names (see Table 3.4, “Schema Namespace Constants”) which are provided as constants. When specifying a property name, you can specify a namespace prefix only when a nested property is defined in a namespace other than the parent property. This can happen when a property has a structured value.

Figure 3.1 shows a diagram of a MetaXAP tree. Properties are organized by schema name. Each property can be accessed with a path string, as described below.
3.3 Path Composition

The MetaXAP object contains one or more trees that represent the metadata properties. Any value (leaf node) can be directly accessed by composing a path to the value using a string...
The path notation is modelled on the XPath standard, but uses a very narrow subset of the standard. This means that paths that are valid for MetaXAP are also valid in a general XPath implementation. The converse is not true: general XPath expressions are not necessarily valid paths for MetaXAP.

The paths specified to the MetaXAP object are all relative to an implicit document root. The path for the Name property is “Name”, not “/Name”.

The paths are not literal paths that match the RDF representation exactly. For one thing, there are multiple RDF serializations which generate the same abstract tree. Paths are normalized to the simplified representation exemplified by the diagram above. When in doubt, use paths that are returned by the enumerate functions.

The most obvious consequence of this is that when referring to structured containers, the actual element that represents each item, rdf:li, is elided. This means that all items of a container are referred to with a wild card in place of the rdf:li item, e.g., “title/*[1]”, is the first title alternative.

Here are some examples which are based on the diagram in Figure 3.1.

The paths to all of the values (leaf nodes) associated with the “http://ns.adobe.com/xap/1.0/” namespace (XAP_NS_XAP), and the values as returned are:

<table>
<thead>
<tr>
<th>Path</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>John Smith</td>
</tr>
<tr>
<td>Keywords/*[1]</td>
<td>API</td>
</tr>
<tr>
<td>Keywords/*[2]</td>
<td>metadata</td>
</tr>
</tbody>
</table>

The paths to all of the values (leaf nodes) associated with the “http://purl.org/dc/elements/1.0/” namespace (XAP_NS_DC), and their values, are:

<table>
<thead>
<tr>
<th>Path</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>title/*[@xml:lang='en-us']</td>
<td>The XMP Toolkit Specification</td>
</tr>
<tr>
<td>title/*[@xml:lang='it']</td>
<td>La Specifica Di XMP Toolkit</td>
</tr>
</tbody>
</table>

For containers, you may use the “last()” function to specify the last item in the container, whatever it may be. So, for example, the Italian alternative of the title can be found at “title[last()]”.

Also, you can use ordinal numbers to select items in a container. The first item is “1”. Thus, the English version of the title can be accessed with the path “title/*[1]”.

<table>
<thead>
<tr>
<th>Path</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>title/*[@xml:lang='en-us']</td>
<td>The XMP Toolkit Specification</td>
</tr>
<tr>
<td>title/*[@xml:lang='it']</td>
<td>La Specifica Di XMP Toolkit</td>
</tr>
</tbody>
</table>
The paths to all of the values (leaf nodes) associated with the “http://ns.adobe.com/xap/1.0/s/” namespace (XAP_NS_XAP_S), and their values, are:

<table>
<thead>
<tr>
<th>Path</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileDisposition/*[1]/os</td>
<td>URL</td>
</tr>
<tr>
<td>FileDisposition/*[1]/path</td>
<td><a href="http://atg/projects/xmp/">http://atg/projects/xmp/</a></td>
</tr>
<tr>
<td>FileDisposition/*[1]/name</td>
<td>xapi.html</td>
</tr>
</tbody>
</table>

In most cases, the path is specified all the way to a leaf node, but in some cases, it is useful to specify an intermediate node, such as for the count method below. Simply compose the path to the name of the node, and use the appropriate count terminator (‘*’ for element children). For example, to count the number of items that the title container has, pass the “title/*” path.

### 3.3.1 XPath Syntax

A MetaXAP object contains an XML tree. Any node can be accessed by composing a path to the node. These paths can be simply encoded in a string. You cannot use a fully general XPath in the XMP Toolkit. You must use paths that conform to the very narrow subset specified below.

The path notation is modelled on the XPath standard, but uses a very narrow subset of the standard. This means that paths that are valid for MetaXAP are also valid in a general XPath implementation. The converse is not true: general XPath expressions are not necessarily valid paths for MetaXAP.

The following is a complete BNF of the path composition grammar:

```
path ::= QName | path '/' expr
Qname ::= name | name ':' name
expr ::= QName | '*[' pred ']
pred ::= ordinal | 'last()' | QName '=' literal | '@xml:lang=' literal
```

No productions are given for ordinal, name, or literal. An ordinal is any positive, non-zero decimal integer. A name is a non-qualified name (NCName) from the XML namespace grammar. Basically, a name consists of a letter or underscore followed by zero or more letters, digits, underscores, hyphens, or periods (for more details, see [http://www.w3.org/TR/REC-xml-names](http://www.w3.org/TR/REC-xml-names)).

A literal is a normal XML quoted string; that is it is surrounded with quotes (") or apostrophes (’) and does not contain the quoting character. If it is necessary to use a quote or apostrophe in a literal, use the HTML character entity names “&quot;” or “&apos;”, respectively (that is, using character entities as escaped versions of those characters).

There are implied prefixes and functions to the path. The implied prefix is derived from the context of the tree. Paths are always relative to that context, and begin with a child of the
document node. The implied function for element and attribute leaf nodes is “text()”, which matches all text node children of the current context node (as specified in the full XPath grammar, but not in this subset). See the member function descriptions and derived classes for additional context implications.

Here is an example of a simple RDF tree that we’ll use to illustrate the syntax:

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
  <rdf:Description about='' xmlns:ex='http://ns.adobe.com/ex/0.0/'>
    <ex:simple>0</ex:simple>
    <ex:struct rdf:parseType='Resource'>
      <ex:a>1</ex:a>
      <ex:b>2</ex:b>
    </ex:struct>
    <ex:set>
      <rdf:Bag>
        <rdf:li rdf:parseType='Resource'>
          <ex:a>3</ex:a>
          <ex:b>4</ex:b>
        </rdf:li>
        <rdf:li rdf:parseType='Resource'>
          <ex:a>5</ex:a>
          <ex:b>6</ex:b>
        </rdf:li>
      </rdf:Bag>
    </ex:set>
    <ex:text xml:lang='en'>English text.</ex:text>
    <ex:one-of>
      <rdf:Alt>
        <rdf:li xml:lang='en-us'>trunk</rdf:li>
        <rdf:li xml:lang='en-gb'>boot</rdf:li>
      </rdf:Alt>
    </ex:one-of>
  </rdf:Description>
</rdf:RDF>
```

The paths to all of the leaf nodes in the RDF example given above, are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Path</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>0</td>
</tr>
<tr>
<td>struct/a</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 3.2 lists the features that modify the getting and setting of property values.

**Table 3.2 Property Value Feature Bits.**

<table>
<thead>
<tr>
<th>Feature Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAP_FEATURE_NONE</td>
<td>No features, value is literal text.</td>
</tr>
<tr>
<td>XAP_FEATURE_XML</td>
<td>Value should be interpreted as XML. Example, “&lt;DOC&gt;Text&lt;/DOC&gt;”. When setting the property, your raw XML is converted by MetaXAP into literal text, with appropriate character entities for parsing characters. The property is stored using an rdf:value, and qualified with iX:iS, whose value is “XML”.</td>
</tr>
<tr>
<td>XAP_FEATURE_RDF_RESOURCE</td>
<td>Value is a URI stored as an rdf:resource.</td>
</tr>
<tr>
<td>XAP_FEATURE_RDF_VALUE</td>
<td>Value is stored with an rdf:value. This bit is not set for XAP_FEATURE_XML, even though it uses rdf:value.</td>
</tr>
</tbody>
</table>

All features bits are mutually exclusive except that XAP_RDF_RESOURCE can be combined with XAP_RDF_VALUE.
3.5 Standard Attributes

Only one standard attribute is supported, the `xml:lang` attribute.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Usage</th>
</tr>
</thead>
</table>

3.6 MetaXAP Class

XMP metadata is a document-ordered collection of RDF description objects. These description objects are parsed and normalized. Properties in the description objects are organized by their schema name.

3.6.1 Storage Management

MetaXAP uses standard `<malloc.h>` and `<new>` allocators. These allocators may be overridden at XMP Toolkit compile time by defining the `XAP_CUSTOM_ALLOC` definitions, and providing an implementation `xap_custom_alloc.h` file. See `XAPTkAlloc.h` for more details.

All data passed to MetaXAP is copied. All data returned from MetaXAP is a copy that the client is responsible for freeing. When the MetaXAP class is destroyed, all of its internally allocated memory is freed.

In order to allow for flexible implementation of internal storage management, clients should know the following:

- MetaXAP to MetaXAP assignment is prohibited.
- The MetaXAP copy constructor is prohibited.
3.7 Important Types Used In MetaXAP

XAPDateTime

typedef struct {
    short sec;          // seconds after the minute - [0,59]
    short min;          // minutes after the hour - [0,59]
    short hour;         // hours since midnight - [0,23]
    short mday;         // day of the month - [1,31]
    short month;        // month of the year - [1,12]
    short year;         // years since 1900 (can be negative!)
    short tzHour;       // hours +ahead/-behind UTC - [-12,12]
    short tzMin;        // minutes offset of UTC - [0,59]
    long nano;          // nanoseconds after second (if supported)
    long seq;           // sequence number (if nano not supported)
} XAPDateTime;

This structure is used to represent dates and times from metadata, and timestamps for media
management and metadata merging. If the system clock used for time is capable of sub-second
resolution, the \texttt{nano} field can be used to represent the sub-second value. If the system clock is
not capable of sub-second resolution, the \texttt{seq} field should be used to guarantee unique
timestamps. If \texttt{seq} is zero, the \texttt{nano} field contains a valid sub-second value. See
\texttt{MetaXAP::XAPClock} below.

MetaXAP::XAPClock

class XAPClock {
   public:
      virtual void
timestamp ( XAPDateTime& dt ) = 0;
   protected:
      virtual ~XAPClock() {};
};

\textit{Description}

Clients provide the clock used for creating timestamps. MetaXAP will never try to delete a
XAPClock object.

Even though the XAPDateTime data structure includes time zone information, XAPClock
should only generate GMT (UTC) timestamps. Code that uses \texttt{MetaXAP::XAPClock} will
check to make sure that the \texttt{tzHour} and \texttt{tzMin} fields are zero. If either is not, a
\texttt{xap_bad_number} exception will be thrown.

The \texttt{seq} field of XAPDateTime allows flexible implementation of the timestamp function.
Consider an implementation based on a system clock that only guarantees time resolution to
the second. Since it is likely that metadata changes will happen in far less than a second, an
implementation like the following could be used:
class MyXAPClock : public XAPClock {
    public:
        long m_seq;              // Internal counter
        struct tm m_last;       // Last timestamp
...
    virtual void
        timestamp ( XAPDateTime& dt ) {
            struct tm now = sysclock(); // 1-second resolution
            if (/* ... now == m_last ... */) {
                dt.seq = ++m_seq;
            } else {
                m_last = now;
                m_seq = 1;
            }
            /* ... convert now to XAPDateTime, and assign to dt ... */
            dt.seq = m_seq;
            dt.nano = 0; // We are using seq
        }
    }
};

Note that the seq field is initialized to 1. The value 0 for seq is reserved to indicate that the nano field should be used instead. If seq is non-zero, nano should be set to 0.

If the system clock has better than second resolution, to the extent that consecutive calls to timestamp will never result in the same time, the nano field can be set to the sub-second value instead, and seq should be set to 0.

MetaXAP::XAPChangeBits

typedef long int XAPChangeBits;

Description

Each timestamp record includes an indication of how the property was last changed. Only one bit is set for any given record, except that XAP_CHANGE_SUSPECT may also be set for any record. This means that only the most recent change is ever recorded. Each bit is described in Table 3.3.
### Table 3.3 XAP Change Bits

<table>
<thead>
<tr>
<th>Change Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAP_CHANGE_NONE</td>
<td>No change bits are set.</td>
</tr>
<tr>
<td>XAP_CHANGE_CREATED</td>
<td>Property was created (defined).</td>
</tr>
<tr>
<td>XAP_CHANGE_SET</td>
<td>Property value was set.</td>
</tr>
<tr>
<td>XAP_CHANGE_REMOVED</td>
<td>Property was removed (undefined)</td>
</tr>
<tr>
<td>XAP_CHANGE_FORCED</td>
<td>The timestamp for this property was forced to a specified value.</td>
</tr>
<tr>
<td>XAP_CHANGE_SUSPECT</td>
<td>There is reason to believe that the timestamp record is invalid.</td>
</tr>
</tbody>
</table>

#### 3.7.1 Namespace Constants

Use these namespace constants for the specified schema descriptions.

### Table 3.4 Schema Namespace Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Schema Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAP_NS_XAP</td>
<td>XMP Core Schema</td>
</tr>
<tr>
<td>XAP_NS_XAP_G</td>
<td>XMP Graphics</td>
</tr>
<tr>
<td>XAP_NS_XAP_G_IMG</td>
<td>XMP Graphics: Image</td>
</tr>
<tr>
<td>XAP_NS_XAP_DYN</td>
<td>XMP Dynamic Media</td>
</tr>
<tr>
<td>XAP_NS_XAP_DYN_A</td>
<td>XMP Dynamic Media: Audio</td>
</tr>
<tr>
<td>XAP_NS_XAP_DYN_V</td>
<td>XMP Dynamic Media: Video</td>
</tr>
<tr>
<td>XAP_NS_XAP_T</td>
<td>XMP Text</td>
</tr>
<tr>
<td>XAP_NS_XAP_T_PG</td>
<td>XMP Text: Paged Text</td>
</tr>
<tr>
<td>XAP_NS_XAP_RIGHTS</td>
<td>XMP Rights Management</td>
</tr>
<tr>
<td>XAP_NS_XAP_MM</td>
<td>XMP Media Management</td>
</tr>
<tr>
<td>XAP_NS_XAP_S</td>
<td>XMP Support</td>
</tr>
<tr>
<td>XAP_NS_XAP_BJ</td>
<td>XMP Basic Job Ticket</td>
</tr>
<tr>
<td>XAP_NS_PDF</td>
<td>Adobe PDF</td>
</tr>
</tbody>
</table>
### 3.8 MetaXAP Member Functions

**public default constructor**

```cpp
MetaXAP ();
```

**Description**

Creates an empty object with no clock.

**public construct empty with clock**

```cpp
MetaXAP ( XAPClock* clock );
```

**Description**

Creates an empty object with a clock. If the `XAP_OPTION_AUTO_TRACK` option is enabled, timestamps will be kept per-property for all changes, and the `xap:MetadataDate` will be set to the last modified time of any change.

**Exceptions**

- `bad_alloc`, `xap_bad_number`

The clock must not be NULL. This constructor will test the clock implementation to make sure it generates GMT (UTC) time (the timezone fields `tzHour` and `tzMin` must both be zero). If this test fails, this constructor will throw a `xap_bad_number` exception.
public construct from buffer

MetaXAP (const char* xmlbuf,
    const long int len,
    const long int opt = XAP_OPTION_DEFAULT,
    XAPClock* clock = NULL);

Description

Constructs a populated MetaXAP from a single buffer of raw XML. The buffer is fed into an XML parser, and the MetaXAP is populated with sub-objects. If there are multiple buffers, use the default constructor instead and call parse.

The specified options opt are enabled immediately after the empty MetaXAP instance is created. This is particularly useful for enabling auto-tracking to capture creation dates for properties as they are parsed (assuming they don’t already have timestamps).

If clock is NULL, no automatic tracking is done. Either the client does it manually with the get/set timestamp functions (listed below), or no timestamps are generated for this metadata.

MetaXAP destructor

virtual ~MetaXAP ();

Description

Destroy this object and all internally allocated memory.

MetaXAP::append

typedef long int XAPFeatures;
virtual void append (const std::string& ns,
    const std::string& path,
    const std::string& value,
    const bool inFront = false,
    const XAPFeatures f = XAP_FEATURE_DEFAULT);

Description

Creates a new property with the specified value, and adds it next to the property specified by namespace ns and path. The path must specify a property in a structured container. The inFront parameter says whether to place the new value before or after the named position. To add a property to the end of a container, use the “last()” specifier, for example, “title/*[last()]”. To add a property or attribute to the beginning of a container or list of attributes, use the pattern “*[1]” in the path and pass TRUE for inFront.
The `append` function is not supported for attributes.

**Examples**

```
m.append ( XAP_NS_XAP, "FileDisposition/*[last()]/os", "URL" );
m.append ( XAP_NS_XAP, "title/*[1]", "First Title", true );
```

All properties related to the specified property by alias or actual value that are also containers are appended as well (see `MetaXAP::SetAlias`). For example, suppose `Car` is an alias of `Vehicle`, and `Auto` is an alias of `Vehicle`. If any of `Car`, `Auto`, or `Vehicle` is appended, all that are containers are appended as well.

**Exceptions**

`bad_alloc`, `xap_bad_path`, `xap_bad_type`, `xap_bad_number`, `xap_bad_schema`

 Throws exceptions for syntactically invalid paths, and for attempting to append to a property that is not a structured container. Throws `xap_bad_number` if the specified ordinal is beyond “last()”. Throws `xap_bad_schema` if `ns` is not registered or invalid.

**MetaXAP::count**

```
virtual size_t count ( const std::string& ns,
                     const std::string& path ) const;
```

**Description**

Returns the number of items in the structured container specified by `ns` and `path`.

**Example**

```
size_t n = m.count ( XAP_NS_DC, "title/*" );  // number of language alts
```

**Exceptions**

`bad_alloc`, `xap_bad_path`, `xap_bad_schema`

Throws `xap_bad_path` for syntactically invalid paths, or if the path does not end with “*”. Throws `xap_bad_schema` if `ns` is not registered or invalid.
MetaXAP::createFirstItem

There are two variations:

Variation #1:

typedef long int XAPFeatures;
virtual void
createFirstItem ( const std::string& ns,
        const std::string& path,
        const std::string& value,
        const XAPStructContainerType type = xap_bag,
        const XAPFeatures f = XAP_FEATURE_DEFAULT );

Description

Creates a structured container of the specified type, and set the value of the first item at the end of the specified path, with the optionally specified features. Nodes are created as needed to ensure that the path is complete. See next variation for examples and exceptions.

Variation #2:

virtual void
createFirstItem ( const std::string& ns,
        const std::string& path,
        const std::string& value,
        const std::string& selectorName,
        const std::string& selectorVal,
        const bool isAttr = true,
        const XAPFeatures f = XAP_FEATURE_DEFAULT );

Description

Creates a structured container of the type xap_alt, and set the value of the first item at the end of the specified path, with the specified selectorName and selectorVal as the selector of the alternation, and optional features. Expressed as an XPath predicate, the selector would be [@selectorName='selectorVal'] if the isAttr is TRUE, otherwise it would be “[selectorName='selectorVal’]” and value is ignored (just pass selectorValue or ““). Nodes are created as needed to ensure that the path is complete.

All properties related to the specified property by alias or actual value that are also containers are created as well (see MetaXAP::SetAlias). For example, suppose Car is an alias of Vehicle, and Auto is an alias of Vehicle. If any of Car, Auto, or Vehicle does not exist and is a container type, each is created (nothing happens to any that do exist, or are not containers).
**Examples**

```c++
//Create the first keyword
m.createFirstItem ( XAP_NS_XAP, "Keywords", "big" );

//Create the first Title, selected by xml:lang of en-us
//The path to get this item would be "Title/*[@xml:lang='en-us']"
 m.createFirstItem ( 
    XAP_NS_XAP, "Title", "Your Photo", "xml:lang", "en-us" );

//Create the first FileDisposition, selected by sub-prop os of UNIX
//The path to get this item would be "FileDisposition/*[os='UNIX']"
 m.createFirstItem ( 
    XAP_NS_XAP_S, "FileDisposition", "", "os", "UNIX", false );
```

**Exceptions**

- bad_alloc, xap_bad_path, xap_bad_type, xap_bad_schema

Throws xap_bad_path for syntactically invalid paths and for a path that leads to a property that is already defined. Use **MetaXAP::append** to add additional items to the container.

Throws xap_bad_schema if ns is not registered or invalid. Throws xap_bad_type if not a container.

**MetaXAP::enable**

```c++
typedef long int Options;
virtual void
enable ( const Options opt,
            const bool en ) throw ();
```

**Description**

Enables or disables the specified option(s), such as XAP_OPTION_DEBUG. Unrecognized options are ignored.

The options are defined in Table 3.5, “Option Enable Constants.”
### Table 3.5 Option Enable Constants

<table>
<thead>
<tr>
<th>Option</th>
<th>When option is enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAP_OPTION_NONE</td>
<td>No options.</td>
</tr>
<tr>
<td>XAPOPTION_DEFAULT</td>
<td>Default options in force.</td>
</tr>
<tr>
<td>XAP_OPTION_ALIASING_ON</td>
<td>Alias mapping occurs during property get, set, etc., (see <code>MetaXAP::SetAlias</code>). If disabled, property get, set, etc., occurs on the specified property only. Enabled by default.</td>
</tr>
<tr>
<td>XAPOPTION_ALIAS_OUTPUT</td>
<td>If enabled, all forms of aliased properties are written when serializing. Otherwise only the base form of each alias set is written. Disabled by default.</td>
</tr>
<tr>
<td>XAP_OPTION_AUTO_TRACK</td>
<td>When constructed with a XAPClock object, automatically modify <code>xap</code> metadata properties for media management that pertain to this metadata instance. For example, calls to <code>set</code> will cause the <code>xap:MetadataDate</code> and per-property timestamps to be updated. See setup below. Enabled by default.</td>
</tr>
<tr>
<td>XAP_OPTION_DEBUG</td>
<td>Pre- and post-condition checking and other assertions are activated for the debug version of the Toolkit only. Disabled by default.</td>
</tr>
<tr>
<td>XAP_OPTION_XAPMETA_ONLY</td>
<td>If enabled, the parser will only recognize RDF elements that are descendents of the tag “xapmeta” in XAP_NS_META namespace. If disabled, the parser will recognize all RDF elements, regardless of their location in the XML document. See parse for more details. Enabled by default.</td>
</tr>
<tr>
<td>XAP_OPTION_XAPMETA_OUTPUT</td>
<td>A xapmeta element in the XAP_NS_META namespace is written as the outermost XML element when serializing. Enabled by default.</td>
</tr>
</tbody>
</table>

**Example**

```c++
meta->enable ( XAP_OPTION_TAG_ONLY, false );
```
MetaXAP::enumerate

There are three variations:

**Variation #1:**

```cpp
virtual XAPPaths*
enumerate ( const int depth = 0 );
```

**Description**

Returns a pointer to an object that enumerates properties in this MetaXAP object. Properties
are listed in document order, or the order in which they were specified. Attributes are always
listed before child properties. It is the responsibility of the caller to destroy the XAPPaths
object. Changes to MetaXAP (calls to non-const member functions) are not reflected in the
XAPPaths object.

The `depth` parameter limits the depth of the enumeration. If the value is 0 (default), paths to
all leaf nodes are enumerated, regardless of the number of steps to each leaf. If the value is 1,
only the paths with one step (no slash) are generated, which correspond to the top-level nodes
of the tree. If the value is 2, paths that only have two steps (one slash) or less, and generally
include the attributes of top-level nodes if any, and children of top-level nodes, if any. And so
on.

**Example**

```cpp
string ns, prop, val;
XAPFeatures f;

XAPPaths* p = m->enumerate();
while ( p->hasMorePaths() ) {
    p->nextPath ( ns, prop );
}
if ( m->get ( ns, prop, val, f ) ) {
    cout << prop << "=" << val << endl;
}
delete p;
delete m;
```

**Exceptions**

`bad_alloc`

**Variation #2:**

```cpp
virtual XAPPaths*
enumerate ( const std::string& ns,
            const std::string& subPath,
            const int steps = 0 );
```
**MetaXAP**

*MetaXAP Member Functions*

---

**Description**

Returns a pointer to an object that enumerates all of the properties in the specified `subPath`. Children are listed in the order they are specified, and attributes are always listed before child properties. It is the responsibility of the caller to destroy the XAPPaths object. Changes to MetaXAP (calls to non-const member functions) are not reflected in the XAPPaths object. The steps parameter is described above.

**Example**

```c++
string ns, path, val;
XAPFeatures f;

XAPPaths* p = m->enumerate(XAP_NS_XAP, "TestCont");
while ( p->hasMorePaths() ) {
    p->nextPath ( ns, path );
    if ( m->get ( ns, prop, val, f ) ) {
        cout << prop << "=" << val << endl;
    }
    delete p;
    delete m;
}
```

**Exceptions**

- `bad_alloc`, `xap_bad_path`, `xap_bad_schema`

Throws `xap_bad_schema` if `ns` is invalid. Throws `xap_bad_path` if the path is invalid.

**Variation #3:**

```c++
typedef enum {
    xap_before,
    xap_at,
    xap_after,
    xap_noTime,
    xap_notDef
} XAPTimeRelOp;

virtual XAPPaths*
enumerate ( const XAPTimeRelOp op,
    const XAPDateTime& dt,
    const XAPChangeBits how = XAP_CHANGE_MASK );
```

**Description**

Returns a pointer to an object that enumerates all of the properties whose last modified timestamp has the relation to `dt` specified by `op`. For example, if `dt` has an earlier time than the timestamp for “Foo” (i.e., “Foo” is newer than whatever `dt` specifies), “Foo” would be
included in the enumeration if op is xap_after, and would not be included if the op is xap_at or xap_before. Returns NULL if there are no matches.

The op xap_noTime matches any property that does not have a timestamp. The op xap_notDef is ignored. The bits set in how act as a filter against which properties are included in the comparison with op. For example, to enumerate only those properties that have been removed since dt:

```cpp
... = meta->enumerate ( xap_after, dt, XAP_CHANGE_REMOVED );
```

**Exceptions**

bad_alloc

## MetaXAP::extractSerialization

```cpp
virtual size_t
extractSerialization ( char* buf,
const size_t nmax );
```

Call `extractSerialization` to incrementally extract the contents of the string saved by a preceding call to `serialize`. You specify the size of your buffer with parameter `nmax`. The function returns the number of bytes (char) that were actually copied. When the function returns 0, the extraction is complete and the private string is emptied. Subsequent calls to `extractSerialization` will result in no copies and a return value of 0, until `serialize` is called again.

**Example**

```cpp
const int bufMetaMax = 1024;
char bufMeta[max];
(void) = meta->serialize ( xap_format_pretty, 0 );
while (true) {
    if ( size == 0 ) break;
    szz = meta->extractSerialization ( bufMeta, bufMetaMax - 1 );
    cout->write ( bufMeta, szz );
}
```
MetaXAP::get

typedef long int XAPFeatures;
virtual bool
going { const std::string& ns,
    const std::string& path,
    std::string& val,
    XAPFeatures& f ) const;

Description

Gets the value at the property specified by ns and path as a string. If any node along the path does not exist, get returns FALSE, otherwise it returns TRUE and the string value is copied into val. The features of the string value, such as whether or not XML markup is preserved, are copied into f.

Example

    bool is;
    XAPFeatures f;
    std::string v;
    is = m.get ( XAP_NS_XAP, "Nickname", v, f );
    is = m.get ( XAP_NS_DC, "title/*[@xml:lang='it']", v, f );

Exceptions

    bad_alloc, xap_bad_path, xap_no_match, xap_bad_schema

Throws exceptions for syntactically invalid paths, and paths that do not match any property (such as trying to get item 5 from an existing simple value). Throws xap_bad_schema if ns is not registered or invalid.

MetaXAP::getContainerType

typedef enum {
    xap_alt,
    xap_bag,
    xap_seq,
    xap_sct_unknown
} XAPStructContainerType;

virtual XAPStructContainerType
getContainerType ( const std::string& ns,
    const std::string& path ) const;
**Description**

Returns the type of the specified container. The path must specify a container type (`MetaXAP::getForm` must return `xap_container`).

**Examples**

```cpp
XAPStructContainerType t =
    m.getContainerType ( XAP_NS_XAP, "FileDisposition" );
```

**Exceptions**

`bad_alloc`, `xap_bad_path`, `xap_no_match`, `xap_bad_schema`

Throws `xap_bad_schema` if `ns` is not registered or invalid. Throws `xap_bad_path` if the path is invalid. Throws `xap_no_match` if the path is syntactically valid, but does not match any defined property.

**MetaXAP::getForm**

```cpp
typedef enum {
    xap_simple,
    xap_description,
    xap_container,
    xap_unknown
} XAPValForm;
```

```cpp
virtual XAPValForm
getForm ( const std::string& ns,
          const std::string& path ) const;
```

**Description**

Returns the type of property specified by `ns` and `path`, as shown below in Table 3.6.
**TABLE 3.6 Property Type Values**

<table>
<thead>
<tr>
<th>XAPValForm</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xap_simple</td>
<td>Path to a simple value.</td>
</tr>
<tr>
<td>xap_description</td>
<td>Path to a nested description objects. Contains other properties as children.</td>
</tr>
<tr>
<td>xap_container</td>
<td>Path to a structured container. See MetaXAP::getContainerType.</td>
</tr>
<tr>
<td>xap_unknown</td>
<td>Unknown value type (treat as xap_simple with parseType=&quot;Literal&quot;).</td>
</tr>
</tbody>
</table>

**Example**

```cpp
XAPValForm vt = m.getForm ( XAP_NS_XAP, "FileDisposition" );
```

**Exceptions**

bad_alloc, xap_bad_path, xap_no_match, xap_bad_schema

Throws xap_bad_schema if ns is not registered or invalid. Throws xap_bad_path if the path is invalid. Throws xap_no_match if the path is syntactically valid, but does not match any defined property.

**MetaXAP::getResourceRef**

```cpp
virtual void
getResourceRef ( std::string& resRef ) const;
```

**Description**

Returns the reference (URI) for the resource that this MetaXAP is about. Returns the empty string "" if the description is embedded in the resource itself.

**Exceptions**

bad_alloc
**MetaXAP::getTimestamp**

```cpp
template<typename...>
virtual bool
getTimestamp ( const std::string& ns,
                const std::string& path,
                XAPDateTime& dt,
                XAPChangeBits& how ) const;
```

**Description**

Returns *FALSE* if the property specified by `ns` and `path` is not defined. Otherwise, returns *TRUE*, and copies the timestamp value into `dt`. The bits in `how` are set according to how the property was changed. If there is no timestamp record for this property, `how` is set to XAP_CHANGE_NONE.

**Example**

```cpp
XAPDateTime dt;
XAPChangeBits how;
MetaXAP* meta = new MetaXAP();
bool Ok = meta->getTimestamp (XAP_NS_XAP_G_IMG, "Dimensions/stDim:w", dt, how );
```

**Exceptions**

- bad_alloc, xap_bad_path, xap_bad_schema, xap_no_match

Throws exceptions for syntactically invalid paths. Throws xap_bad_schema if `ns` is not registered or invalid. Throws xap_no_match if property is not defined.

---

**MetaXAP::isEnabled**

```cpp
virtual bool
isEnabled ( const Options opt ) const throw ();
```

**Description**

Returns whether the specified option is enabled, such as XAP_OPTION_DEBUG. An unrecognized option always returns *FALSE*. Pass a single option bit.
MetaXAP::parse

```cpp
virtual void
parse ( const char* xmlbuf,
        const size_t n,
        const bool last = false );
```

**Description**

Parses a buffer of XML and creates the corresponding XMP objects. This function expects to be called in the order that buffers occur for a particular XML serialization. The last buffer is indicated by passing `TRUE` for `last`. It is legal for tokens, or even multibyte characters, to cross buffer boundaries.

Only one parsing cycle should be used per MetaXAP instance (a cycle is 0 or more calls to `parse` with `last==false`, I call to `parse` with `last==true`). Calling `parse` with `last==false` after calling it with `last==true` for the same MetaXAP instance will have unspecified results.

The `parse` function will handle any well-formed XML, and will detect RDF elements anywhere in the XML. If the `XAP_OPTION_XAPMETA_ONLY` option is enabled, only those RDF elements that are children of the “xapmeta” tag in the `XAP_NS_META` namespace are recognized as XMP metadata, all others are ignored. If the `XAP_OPTION_XAPMETA_ONLY` option is disabled, all RDF elements in the input are recognized as XMP metadata.

Calling any other functions in MetaXAP during a `parse` will yield undefined results.

**Example**

```cpp
const int bufMetaMax = 1024;
char bufMeta[bufMetaMax];
MetaXAP* meta = new MetaXAP();
ifstream* metaFs = new ifstream ( "metadata.xml",
    ios_base::in | ios_base::binary);
if ( !metaFs || metaFs->fail() ) exit(-1);
try {
    while ( !metaFs->eof() ) {
        metaFs->read ( bufMeta, nbufMetaMax
        meta->parse ( bufMeta, metaFs->gcount() );
    }
    meta->parse ( "\n", 1, true ); // all done
}
catch ( xap_bad_xml& x ) {
    cerr << x.what() << "(" << x.getContext() << ")":"
      << x.getLine() << endl;
    throw;
}````
Exceptions

bad_alloc, xap_bad_xml, xap_bad_xap

Throws xap_bad_xml if the XML is not well-formed (lexical error). Throws xap_bad_xap if the RDF is invalid (parsing error).

MetaXAP::purgeTimestamps

virtual void
purgeTimestamps ( const XAPChangeBits how = XAP_CHANGE_REMOVED,
const XAPDateTime* dt = NULL );

Description

Purges all timestamp records for properties with any XAPChangeBits set in how. By default, purges all timestamp records for properties marked XAP_CHANGE_REMOVED. If dt is not NULL, all timestamps that were not purged are forced to the specified timestamp, and their XAPChangeBits are set to XAP_CHANGE_FORCED. Thus, to force all timestamps to a specific time, pass XAP_CHANGE_NONE as the first parameter and a non-null date and time as the second parameter.

Example

XAPDateTime dt;
meta->purgeTimestamps ( XAP_CHANGE_REMOVED, &dt );

Exceptions

bad_alloc

MetaXAP::remove

virtual void
remove ( const std::string& ns,
const std::string& subPath );

Description

Removes the specified property and all of its sub-properties, if any. When a child of a container is removed, all siblings that follow that item are renumbered. Nothing is done if there is no property for the specified path.

All properties related to the specified property by alias or actual value are removed as well (see MetaXAP::SetAlias). For example, suppose Car is an alias of Vehicle, and Auto is an alias of Vehicle. If any of Car, Auto, or Vehicle is removed, all are removed.
Examples

m.remove ( XAP_NS_DC, "title/*[1]");

Throws an exception if the path is invalid, or the path matches none of the nodes.

Exceptions

xap_bad_path, xap_no_match, xap_bad_schema

Throws xap_bad_schema if ns is not registered or invalid.

MetaXAP::serialize

typedef enum {  
    xap_format_pretty,  
    xap_format_compact
} XAPFormatType;

const int XAP_ESCAPE_CR = 1<<0;
const int XAP_ESCAPE_LF = 1<<1;

virtual size_t serialize ( const XAPFormatType f = xap_format_pretty,  
                           const int escnl = 0 ) = 0;

Description

Serializes the MetaXAP tree as XML. Call serialize to perform serialization, optionally specifying a format and escnl for filtering linebreaks. The f option xap_format_pretty is pretty-printed for human readability, using whitespace and indenting. The f option xap_format_compact minimizes whitespace and uses the most compact representation possible. The serialized data is kept in a private string.

The escnl bits indicate whether line ending characters should be escaped into character refs, using the HTML character entity names "&#xD;" for CR, and "&#xA;" for LF. This allows a client to post-filter the XML to impose line-length limitations: the unescaped version of the line-break character can be inserted into the XML, since the XML is guaranteed not to contain that character unescaped, unless formatted pretty (see below). A processing instruction is added at the beginning to indicate that the filtering was applied. The processing instruction is omitted if escnl is 0. This instruction is detected by the parse function of this class, and the corresponding unescaped linebreak characters, if any, are removed before buffers are passed to the XML parser. If f is xap_format_pretty, lines are formatted with a linebreak character as follows: CR if escnl is XAP_ESCAPE_LF only, LF if escnl is XAP_ESCAPE_CR only, CRLF if both bits are set. Returns a value of 0 if there is no metadata, and a value greater than zero (>0) otherwise.
The serialized metadata does not include (does not begin with) the standard xml prolog `<xml ...?>`. This makes it easier to embed the serialized metadata in an existing XML document entity, such as a WebDAV property. If you are writing this serialized XML as a document entity (e.g., into a standalone file), you should prepend an appropriate prolog, such as:

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

If the `XAP_OPTION_XAPMETA_OUTPUT` option is enabled, the serialized output is contained within the single tag “xapmeta” in the `XAP_NS_META` namespace. If the `XAP_OPTION_XAPMETA_OUTPUT` option is disabled, the “xapmeta” tag is omitted. In either case, all of the metadata is contained within a single RDF element.

The serialized metadata is in UTF-8 Unicode character encoding.

**Exceptions**

bad_alloc

---

### MetaXAP::set

typedef long int XAPFeatures;
virtual void
set ( const std::string& ns,
     const std::string& path,
     const std::string& value,
     XAPFeatures f = XAP_FEATURE_DEFAULT );

**Description**

Sets the specified value at the end of the specified path, with the optionally specified features. Nodes are created as needed to ensure that the path is complete, except for items of a structured container (see MetaXAP::createFirstItem above and xap_bad_number below). Existing values are overwritten.

**Examples**

```c++
    m.set ( XAP_NS_XAP, "Author", "Your Name" );
    m.set ( XAP_NS_XAP_G_IMG, "Dimensions/stDim:w", "480" );
```

All properties related to the specified property by alias or actual value are set as well (see MetaXAP::SetAlias). For example, suppose `Car` is an alias of `Vehicle`, and `Auto` is an alias of `Vehicle`. If any of `Car`, `Auto`, or `Vehicle` is set, all are set to the same value.

**Exceptions**

bad_alloc, xap_bad_path, xap_bad_type, xap_bad_number, xap_bad_schema

Throws exceptions for syntactically invalid paths, and for attempting to change the type of the property, e.g., if “title” is a structured container (an Alt of different languages), trying to set
title to a simple value will generate a xap_bad_type exception. Throws a xap_bad_number exception if an attempt is made to set a structured item beyond “last ()”. Use MetaXAP::append to add items to a container. Throws xap_bad_schema if ns is not registered or invalid.

MetaXAP::setTimestamp

virtual void
setTimestamp ( const std::string& ns,
               const std::string& path,
               const XAPDateTime& dt );

Description
This should only be used when manual tracking is being done by the client. Sets the timestamp to dt. The XAPChangeBits for this property are set to XAP_CHANGE_FORCED.

Example
meta->setTimestamp ( XAP_NS_XAP_G_IMG, "Dimensions/stDim:w", dt );

Exceptions
bad_alloc, xap_bad_path, xap_bad_schema, xap_no_match

The timestamp must be GMT (UTC) time (the timezone fields tzHour and tzMin must both be zero). If there is non-zero timezone information, the xap_bad_number exception will be thrown. Throws xap_bad_path for syntactically invalid paths. Throws xap_no_match for valid paths that have no defined property. Throws xap_bad_schema if ns is not registered or invalid.

MetaXAP::setup

virtual void
setup ( const char *const key,
        const std::string& val );

Description
Some properties require metadata that only the client of this Toolkit can provide, such as the name of the software agent using the Toolkit. Use this function to provide values to this instance of MetaXAP for automatic tracking.
Examples

m.setup ( XAP_SETUP_VENDOR, "Adobe" );
m.setup ( XAP_SETUP_APP, "Photoshop" );
m.setup ( XAP_SETUP_VERSION, "10.0" );
m.setup ( XAP_SETUP_PLATFORM, "Windows" );

These example calls would allow the Toolkit to create an AgentName of “Adobe Photoshop 10.0 for Windows”.

Exceptions
bad_alloc

MetaXAP::setResourceRef

virtual void
setResourceRef ( const std::string& ref );

Description
Sets the reference to the resource (URI) that this MetaXAP is about.

Example

meta->setResourceRef ( "test:/resource/'about'/'" );

Exceptions
bad_alloc

3.9 MetaXAP Static Functions (Class Methods)

MetaXAP::Clone

static MetaXAP*
Clone ( MetaXAP* orig );

Description
Makes a deep-copy of the MetaXAP object orig and returns it. Copies timestamps without changing them, if any.

NOTE: Multi-threaded clients must provide mutual exclusion.
Examples

MetaXAP* clone = MetaXAP::Clone(meta);

MetaXAP::EnumerateAliases

static XAPPaths*
EnumerateAliases () throw();

Description

Returns a pointer to an object that enumerates all of the aliases defined for all MetaXAP objects. It is the responsibility of the caller to destroy the XAPPaths object. Changes to aliases (calls to MetaXAP::SetAlias) are not reflected in the XAPPaths object.

NOTE: Multi-threaded clients must provide mutual exclusion.

MetaXAP::GetAlias

static bool
GetAlias ( const std::string& aliasNS,
const std::string& aliasPath,
std::string& actualNS,
std::string& actualPath,
XAPStructContainerType& cType ) throw();

Description

Gets the alias for the specified path, if any. The first pair, aliasNS and aliasProp, specifies a namespace and path to the property whose actual value might be found elsewhere. If there is an alias defined, actualNS and actualProp are set to the namespace and path, respectively, of the actual property and TRUE is returned. Otherwise, FALSE is returned. The cType is also set to the container type of the actual path: if the value form is not a container, cType is set to xap_sct_unknown.

NOTE: Multi-threaded clients must provide mutual exclusion.

Example

string nsActual, pActual;
string pActual;
XAPStructContainerType sct;
MetaXAP::GetAlias ( XAP_NS_XAP, "TestCont", nsActual, pActual, sct );
MetaXAP::Merge

typedef enum {
    xap_policy_a,
    xap_policy_b,
    xap_policy_newest,
    xap_policy_oldest,
    xap_policy_dont_merge,
    xap_policy_ask_user
} XAPMergePolicy;

static XAPPaths*
Merge ( MetaXAP* a,
    MetaXAP* b,
    MetaXAP** merge,
    const XAPMergePolicy policy,
    const bool justCheck = false,
    XAPTimeRelOp* dontMergeResult = NULL );

Description

If justCheck is FALSE and policy is not xap_policy_dont_merge nor xap_policy_ask_user, this function creates a new MetaXAP object and returns the pointer in merge, after merging the metadata in instance a with instance b, and copying the resulting metadata into merge. Any properties defined in a but not in b, or in b but not in a, are defined (copied) to merge. The corresponding timestamp record is also copied unchanged. The policy specifies what the merge does when both a and b define a property, including cases when one has the XAP_CHANGE_REMOVED bit set. The policy descriptions follow:

**TABLE 3.7 Merge Policy Descriptions**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xap_policy_dont_merge</td>
<td>Just compare, see below.</td>
</tr>
<tr>
<td>xap_policy_a</td>
<td>The value in a is copied to merge.</td>
</tr>
<tr>
<td>xap_policy_b</td>
<td>The value in b is copied to merge.</td>
</tr>
<tr>
<td>xap_policy_newest</td>
<td>The latest timestamped value is copied to merge.</td>
</tr>
<tr>
<td>xap_policy_oldest</td>
<td>The earliest timestamped value is copied to merge.</td>
</tr>
<tr>
<td>xap_policy_ask_user</td>
<td>Same as xap_policy_dont_merge.</td>
</tr>
</tbody>
</table>

Any property with a XAP_CHANGE_SUSPECT bit set is ignored and no change is made to merge for that property, regardless of whether the bit is set in a or b. Properties with no
timestamp are treated as if they had a timestamp equal to the value of xap:MetadataDate. If xap:MetadataDate is not defined, no change is made to merge for that property for xap_policy_newest or xap_policy_oldest only.

The returned paths represent those properties in merge that were changed as a result of the policy, or if justCheck is TRUE, the paths for the properties that would have been copied into merge if justCheck had been FALSE. Does not include properties copied to merge because they were defined in a but not in b, or vice versa. Returns NULL if nothing is copied to merge (merge is unchanged by the call).

For xap_policy_dont_merge and xap_policy_ask_user, no new MetaXAP object is created and merge is left unchanged. If justCheck is FALSE, the paths returned represent those properties that are defined in both a and b, but that do not have identical timestamps. If justCheck is TRUE, NULL is returned, and if dontMergeResult is non-NULL, it is set to the result of comparing the xap:MetadataDate of a and b (see UtilityXAP::CompareTimestamps).

Example

```cpp
MetaXAP* mergedMeta = NULL;

// Merge letting newer values override older values.
XAPPaths* newer = MetaXAP::Merge(oldMeta, newMeta, &mergedMeta,
  xap_policy_newest);

MetaXAP* deltaMeta = NULL;

// Merge letting older values override newer values.
XAPPaths* older = MetaXAP::Merge(oldMeta, newMeta, &deltaMeta,
  xap_policy_oldest);
```

Exceptions

Raises all the same exceptions as MetaXAP::enumerate, MetaXAP::set, MetaXAP::remove, MetaXAP::createFirstItem, and MetaXAP::setTimestamp.

MetaXAP::RegisterNamespace

```cpp
static void
RegisterNamespace (const std::string& nsName,
  const std::string& suggestedPrefix);
```

Description

For serialization to XML, clients must provide a suggested prefix for each namespace that they use. The standard namespaces (those for which a constant string is defined in this API) already have registered prefixes. Register a namespace name (which should be a URI), and a suggested
prefix for composing qualified names. Omit the composition character (such as “:” for RDF) from the prefix. Setting or creating a property in a namespace that has not been registered will result in an exception.

**NOTE:** Multi-threaded clients must provide mutual exclusion.

**Example**

MetaXAP::RegisterNamespace("http://purl.org/dc/qualifiers/1.0/", "dcq");

**Exceptions**

bad_alloc

---

**MetaXAP::RemoveAlias**

```cpp
static void
RemoveAlias ( const std::string& aliasNS,
              const std::string& aliasPath );
```

**Description**

Removes the specified alias from the alias map for all metadata objects. This function does not change any metadata values. See the important **Note** in MetaXAP::SetAlias, which applies to MetaXAP::RemoveAlias as well.

---

**MetaXAP::SetAlias**

```cpp
static void
SetAlias ( const std::string& aliasNS,
           const std::string& aliasPath,
           const std::string& actualNS,
           const std::string& actualPath,
           const XAPStructContainerType cType = xap_sct_unknown );
```

**Description**

Adds to the alias map for all instances of MetaXAP. Matching aliases are overwritten, new aliases are appended. The alias is specified as two pairs of strings. The first pair, aliasNS and aliasProp, specifies a namespace and path to the property whose actual value is found elsewhere. The second pair, actualNS and actualProp, specifies a namespace and path to the property for the actual value. The cType specifies the container type, if the actualPath represents a container or container member.
Examples

/* "Author" and "Title" in the XMP core schema are aliases of "creator" and "title" in the Dublin Core schema. */

const char* XAP_NS_XAP = "http://ns.adobe.com/xap/1.0/";
const char* XAP_NS_DC = "http://purl.org/dc/elements/1.0/";

MetaXAP::SetAlias ( XAP_NS_XAP, "Author", 
                  XAP_NS_DC, "creator/*[1]", xap_bag );
MetaXAP::SetAlias ( XAP_NS_XAP, "Title", 
                  XAP_NS_DC, "title", xap_alt );

To determine which of two properties should be the alias, and which the actual, consider which will be used most frequently by the broadest cross-section of users. If one property happens to be from a broadly supported schema, such as Dublin Core, or if one property represents an important legacy metadata format, such as IPTC, use that property as the actual, and use the new or XMP defined property as the alias.

NOTE: Changes to the alias map made by calls to SetAlias do not automatically take effect on existing MetaXAP instances. For this reason, it is strongly recommended that all aliases be set prior to any MetaXAP objects being created, and then once they are created, no new alias settings are made until all MetaXAP objects have been destroyed.

If this is not feasible, it is possible to force an existing MetaXAP object to recognize new alias settings. For all MetaXAP objects which have the XAP_OPTION_ALIAS_ON enabled, toggle the option: that is, disable it, and then enable it again, as follows:

MetaXAP* meta;
if ( meta->isEnabled ( XAP_OPTION_ALIAS_ON ) ) {
   meta->enable ( XAP_OPTION_ALIAS_ON, false );
   meta->enable ( XAP_OPTION_ALIAS_ON, true );
}

Exceptions

bad_alloc, xap_bad_path

Throws xap_bad_path if an alias loop is defined, or if an attempt is made to make an alias of an alias, or if an attempt is made to use a property that has previously been defined as an actual value as an alias, or if the aliasPath is malformed. Only single level aliases are supported.
3.10 XAPPPaths Class

This is a pure virtual base class, used to represent an enumeration of the paths to nodes of metadata.

**NOTE:** It is up to the caller to destroy this object with the public destructor. Examples for hasMorePaths and nextPath are shown with MetaXAP::enumerate.

**XAPPPaths::hasMorePaths**

```cpp
virtual bool hasMorePaths() const throw () = 0;
```

**Description**

Returns *TRUE* if there are more paths in the enumeration, otherwise returns *FALSE*.

**XAPPPaths::nextPath**

```cpp
virtual void nextPath ( std::string& ns,
                        std::string& path ) = 0;
```

**Description**

Copies the next namespace and path into the parameters. Calling this method after hasMorePaths has returned *FALSE* will cause the parameters to be set to empty strings.
4 UtilityXAP

4.1 UtilityXAP

UtilityXAP is a collection of static (class) functions that provide general purpose convenience routines.

4.2 UtilityXAP Static Functions (Class Methods)

UtilityXAP::cAnalyzeStep

```cpp
static bool
AnalyzeStep ( const std::string& fullPath,
              std::string& parentPath,
              std::string& lastStep,
              long int& ord,
              std::string& selectorName,
              std::string& selectorVal );
```

**Description**

Removes last step from the path, and separates it into component pieces.

From `fullPath`, remove the last step and assign it to `lastStep`, and assign the front part of the path to `parentPath`. If the last step contains a predicate expression with an ordinal (which is always greater than 0), it is assigned to `ord`. If the ordinal predicate is the function `last()`, `ord` is set to 0. Otherwise, `ord` is set to -1. If the predicate is a selector, such as `[@xml:lang='fr']`, `selectorName` would be assigned `@xml:lang` and `selectorVal` would be assigned “fr”. Otherwise, `selectorName` and `selectorVal` are assigned the empty string.

UtilityXAP::CompareTimestamps

```cpp
static XAPTimeRelOp
CompareTimestamps ( MetaXAP* a,
                    MetaXAP* b,
                    const std::string& ns,
                    const std::string& path );
```
**Description**

Compares the property with the specified namespace ns and path in instance a with instance b, and returns the relation as follows:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>(a timestamp earlier than b) xap_before</td>
</tr>
<tr>
<td>a == b</td>
<td>(a timestamp same as b) xap_at</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>(a timestamp later than b) xap_after</td>
</tr>
<tr>
<td>a ? b</td>
<td>(a or b does not have a timestamp) xap_noTime</td>
</tr>
<tr>
<td></td>
<td>(a or b not defined) xap_notDef</td>
</tr>
</tbody>
</table>

**Example**

UtilityXAP::CompareTimestamps ( meta, clone, XAP_NS_XAP, "Number" );

**Exceptions**

Raises all the same exceptions as MetaXAP::enumerate and MetaXAP::getTimestamp, except that xap_no_match is converted into the return value xap_notDef.

**UtilityXAP::CreateXMLPacket**

```cpp
static void
CreateXMLPacket ( const std::string& encoding,
                  const bool inPlaceEditOk,
                  const size_t padBytes,
                  const std::string& nl,
                  std::string& header,
                  std::string& trailer,
                  std::string* xml = NULL );
```

**Description**

Use this routine to compute the header and trailer string for a packet, which you use to create a XMP packet (for information on XMP Packets, see XMP – Extensible Metadata Platform), or if you specify non-NULL XML data, it will also create the entire packet.

If the encoding is empty (""), it defaults to UTF-8. If inPlaceEditOk is TRUE, it marks the packet as okay to edit in-place, otherwise it marks the packet as read-only.
If positive, the `padBytes` parameter specifies the number of bytes of whitespace padding to add to the packet. The padding is placed after the XML data, and before the trailer.

If `padBytes` is negative, its absolute value specifies the length for the completed packet, and the `xml` parameter must be non-`NULL`. The absolute value of `padBytes` must be large enough to contain the complete packet, otherwise `xap_bad_number` is thrown. The appropriate amount of whitespace padding is added to provide the specified total size. This is convenient when formatting a packet to update existing metadata in a file of unknown format.

The `nl` string is the character sequence to use as a newline between the header and the `xml` data if `xml` is non-`NULL`; it can be empty (`""`), or some combination of well-formed XML whitespace. The header is assigned to the string representing the computed header for the packet, and the trailer is assigned to the string representing the computed trailer of the packet.

The characters in `xml` specify the XML data for the packet. The same non-`NULL` parameter `xml` is assigned the complete packet, with header, trailer, and padding added. The value of `encoding` must match the encoding of the XML data, but no checking is done to guarantee that it does match.

**Examples**

(for UTF-8 encodings):

```cpp
string header, trailer;
UtilityXAP::CreateXMLPacket ( "", true, val.size(), "\n", header, trailer, &val );
```

There is a second form:

```cpp
static void CreateXMLPacket ( const std::wstring& encoding, const bool inPlaceEditOk, const size_t padBytes, const std::wstring& nl, std::wstring& header, std::wstring& trailer, std::wstring* xml = NULL );
```

Same as `CreateXMLPacket` above, except that all of the string parameters are 16-bit character strings.

**NOTE:** This function assumes that the XML data is in the native byte order of this machine. It generates packet header text in UCS-2 encoding, with characters in the range U+0000 to U+007F, plus U+FEFF. This refers only to the additional material for the packet wrapper, NOT to the data contents, which are assumed to be XML compatible UCS-2 and are copied unchanged.
Example
(for UTF-16 encodings)

wstring wxml = L"<A foo='1'>
<B>This is some \03a3 16-bit 
text.</B></A>n";
wstring wh;
wstring wt;
UtilityXAP::CreateXMLPacket ( L"UTF-16", false,
    wxml.size()*sizeof(wchar_t), L"n", wh, wt, &wxml );

UtilityXAP::FilterPropPath

static bool
FilterPropPath ( const std::string& tx,
    std::string& propPath );

Description
Filters UI text into valid XPath.
Converts a UTF-8 string tx into a valid XPath, which is also a UTF-8 string propPath. For example, any disallowed characters, like spaces or slashes, or any Unicode characters greater than U+007A, are converted into a series of hexadecimal digits, where every two digits represent a byte of UTF-8. Such sequences are introduced by the character pattern “-_” and closed with “_”. If the original text contains “-_”, it is escaped with “-__”. If the converted character is the initial character, the escape is modified to be “QQ-_”. If such a sequence exists in the original text, it is escaped as “QQ-__”.

For example, if tx is the single Unicode character U+03A3 GREEK CAPITAL LETTER SIGMA in UTF-8 encoding, it is filtered into “QQ-_ce3_”, which represents the two bytes CE and A3 of UTF-8, in hex.

UtilityXAP::GetBoolean

static bool
GetBoolean ( MetaXAP* meta,
    const std::string& ns,
    const std::string& path,
    bool &val );

Description
Gets a property value as a boolean as specified by ns and path. Calls MetaXAP::get. If the property is not defined, returns FALSE. Otherwise, the string value provided by MetaXAP::get is converted into a boolean and copied into val and TRUE is returned.
**Example**

```cpp
bool areYouHappy;
bool ok = UtilityXAP::GetBoolean ( meta, XAP_NS_XAP, "Happy", areYouHappy );
```

**Exceptions**

Raises all the same exceptions as `MetaXAP::get`, plus `xap_bad_xap` if the property value cannot be converted to a boolean.

**UtilityXAP::GetDateTime**

```cpp
static bool GetDateTime ( MetaXAP* meta,
                           const std::string& ns,
                           const std::string& path,
                           XAPDateTime& dateTime );
```

**Description**

Gets a property value as a date and time.

Gets the Date value specified by `ns` and `path`. Calls `MetaXAP::get`. If the property is not defined, it returns `FALSE`. Otherwise, the string value provided by `MetaXAP::get` is converted into values of the `XAPDateTime` record as described below, and timezone offset from GMT, and `TRUE` is returned. If `tzHour` and `tzMin` are both 0, the time returned is UTC (GMT). The `seq` field is always set to 0, and the `nano` field is set to the subsecond time defined in the value of the property, if any. This function implements the Date as specified in XMP – Extensible Metadata Platform; also see ISO 8601: [http://www.w3.org/TR/NOTE-datetime](http://www.w3.org/TR/NOTE-datetime).

**TABLE 4.1 XAPDateTime Field Usage**

<table>
<thead>
<tr>
<th>XAPDateTime field</th>
<th>Usage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>seconds after the minute</td>
<td>[0,59]</td>
</tr>
<tr>
<td>min</td>
<td>minutes after the hour</td>
<td>[0,59]</td>
</tr>
<tr>
<td>hour</td>
<td>hours since midnight</td>
<td>[0,23]</td>
</tr>
<tr>
<td>mday</td>
<td>day of the month</td>
<td>[1,31]</td>
</tr>
<tr>
<td>month</td>
<td>month of the year</td>
<td>[1,12]</td>
</tr>
<tr>
<td>year</td>
<td>year A.D. (can be negative!)</td>
<td></td>
</tr>
</tbody>
</table>
UtilityXAP Static Functions (Class Methods)

<table>
<thead>
<tr>
<th>tzHour</th>
<th>hours +ahead/-behind UTC</th>
<th>[-12,11]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tzMin</td>
<td>minutes offset of UTC</td>
<td>[0,59]</td>
</tr>
<tr>
<td>nano</td>
<td>nanoseconds after second (if supported)</td>
<td></td>
</tr>
<tr>
<td>seq</td>
<td>sequence number (if nano not supported)</td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

(Using HTML format for shorthand):

1994-11-05T08:30-05:00 corresponds to November 5, 1994, 8:30 am, US Eastern Standard Time.

1994-11-05T13:15:30Z corresponds to the same instant.

(C++ code example):

```cpp
XAPDateTime dt;
bool ok = UtilityXAP::GetDateTime(meta, XAP_NS_XAP, "UTC", dt);
```

**Exceptions**

Raises all the same exceptions as `MetaXAP::get`, plus `xap_bad_xap` if the property value cannot be converted to a date and time.

**UtilityXAP::GetInteger**

```cpp
static bool
GetInteger ( MetaXAP* meta,
             const std::string& ns,
             const std::string& path,
             long int &val );
```

**Description**

Gets a property value as an integer.

Gets the integer value specified by `ns` and `path`. Calls `MetaXAP::get`. If the property is not defined, returns `FALSE`. Otherwise, the string value provided by `MetaXAP::get` is converted into an integer and copied into `val`, and `TRUE` is returned.

**Example**

```cpp
long int gNum = sizeof(int);
bool ok = UtilityXAP::GetInteger ( meta, XAP_NS_XAP, "Number",
                                    gNum );
```
Exceptions

Raises all the same exceptions as `MetaXAP::get`, plus `xap_bad_xap` if the property value cannot be converted to an integer.

UtilityXAP::GetReal

```cpp
template bool GetReal ( MetaXAP* meta,
                        const std::string& ns,
                        const std::string& path,
                        double &val );
```

Description

Gets a property value as a real number.

Gets the real (double) value specified by `ns` and `path`. Calls `MetaXAP::get`. If the property is not defined, `FALSE` is returned. Otherwise, the string value provided by `MetaXAP::get` is converted into a real and copied into `val`, and `TRUE` is returned.

Example

```cpp
double gReal;
bool ok = UtilityXAP::GetReal ( meta, XAP_NS_XAP, "Real", gReal );
```

Exceptions

Raises all the same exceptions as `MetaXAP::get`, plus `xap_bad_xap` if the property value cannot be converted to a real.

UtilityXAP::IsAltByLang

```cpp
template bool IsAltByLang ( const XAPPathTree* tree,
                            const std::string& ns,
                            const std::string& path,
                            std::string* langVal = NULL );
```

Description

Returns `TRUE` if the specified `path` evaluates to a member of a structured container that is of type `xap_alt`, and which is selected by the attribute `xml:lang`. If a pointer to a string is passed in `langVal`, the string is assigned with the value of the `xml:lang` attribute.
This function is handy when you are doing an enumerate. If you are searching for a particular language alternative, pass the paths returned by XAPPaths to this function to test for the sought type, and then compare the langVal with the language you seek.

**Exceptions**

Raises all the same exceptions as `MetaXAP::getForm`.

**UtilityXAP::SetBoolean**

```
static void SetBoolean ( MetaXAP* meta,
                        const std::string& ns,
                        const std::string& path,
                        const bool val );
```

**Description**

Sets a property value as a boolean.

Sets the property specified by `ns` and `path` to the specified boolean value. Calls `MetaXAP::set`. Intermediate nodes on the path are created as needed.

**Example**

```cpp
bool happy = true;
UtilityXAP::SetBoolean ( meta, XAP_NS_XAP, "Happy", happy );
```

**Exceptions**

Raises all the same exceptions as `MetaXAP::set`.

**UtilityXAP::SetDateTime**

```
static void SetDateTime ( MetaXAP* meta,
                          const std::string& ns,
                          const std::string& path,
                          const XAPDateTime& dateTime );
```

**Description**

Sets the property value as a date and time.

Sets the property specified by `ns` and `path` to the specified boolean value. Calls `MetaXAP::set`. Intermediate nodes on the path are created as needed.
See **UtilityXAP::GetDateTime** above for the details of usage for `dateTime`. The `seq` and `nano` fields are ignored.

### Example

```cpp
XAPDateTime dt;
UtilityXAP::SetDateTime ( meta, XAP_NS_XAP, "UTC", dt );
```

### Exceptions

Raises all the same exceptions as **MetaXAP::set**.

---

**UtilityXAP::SetInteger**

```cpp
static void SetInteger ( MetaXAP* meta,
                         const std::string& ns,
                         const std::string& path,
                         const long int val );
```

### Description

Sets property value as an integer.

Sets the property specified by `ns` and `path` to the specified integer value. Calls **MetaXAP::set**. Intermediate nodes on the path are created as needed.

### Example

```cpp
long int num = -123456789;
UtilityXAP::SetInteger ( meta, XAP_NS_XAP, "Number", num );
```

### Exceptions

Raises all the same exceptions as **MetaXAP::set**.

---

**UtilityXAP::SetLocalizedText**

```cpp
static void SetLocalizedText ( MetaXAP* meta,
                               const std::string& ns,
                               const std::string& path,
                               const std::string& lang,
                               const std::string& val,
                               const XAPFeatures f = XAP_FEATURE_DEFAULT );
```
**UtilityXAP Static Functions (Class Methods)**

**Description**
Sets the structured container language alternation property specified by ns and path and lang to the text value specified by val and f. Creates the first item of the container if it does not exist, otherwise sets or appends the value as needed. The path should be the path to the container itself, not to any member. See example below.

**Example**
If the desired language alternative is “Title/*[@xml:lang='de'],” pass “Title” as the path, and “de” as the value of lang.

**Exceptions**
Raises all the same exceptions as `MetaXAP::createFirstItem` and `MetaXAP::set`.

**UtilityXAP::SetReal**

```cpp
static void
SetReal ( MetaXAP* meta,
          const std::string& ns,
          const std::string& path,
          const double val );
```

**Description**
Sets a property value as a real.
Sets the property specified by ns and path to the specified real value. Calls `MetaXAP::set`.
Intermediate nodes on the path are created as needed.

**Example**
```cpp
double real = 3.14159012345678;
UtilityXAP::SetReal ( meta, XAP_NS_XAP, "Real", real );
```

**Exceptions**
Raises all the same exceptions as `MetaXAP::set`. 
A.1 Overview

This appendix lists the collection of C++ classes used for exceptions throughout the Toolkit.

A.1.1 Exception Classes

Errors are indicated using exceptions. Member function prototypes use the conventions listed in Table A.1, “XMP Toolkit Exceptions.”

<table>
<thead>
<tr>
<th>Potential Exceptions</th>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>No exceptions possible.</td>
<td>Declared throw ().</td>
</tr>
<tr>
<td>Client violates a pre-condition, or runtime exceptions possible.</td>
<td>Default declaration (no throw clause).</td>
</tr>
</tbody>
</table>

The following are the exceptions for the XMP Toolkit:

```cpp
/* Text messages for standard exceptions. */
extern const char *const XAP_BAD_ALLOC;
extern const char *const XAP_INVALID_ARGUMENT;

/* Text messages for client faults. */
extern const char *const XAP_FAULT_BAD_FEATURE;
extern const char *const XAP_FAULT_BAD_SCHEMA;
extern const char *const XAP_FAULT_BAD_TYPE;
extern const char *const XAP_FAULT_BAD_PATH;
extern const char *const XAP_FAULT_BAD_ACCESS;
extern const char *const XAP_FAULT_BAD_NUMBER;

/* Text messages for XMP errors. */
extern const char *const XAP_ERR_BAD_XAP;
extern const char *const XAP_ERR_BAD_XML;
extern const char *const XAP_ERR_NO_MATCH;

class XAP_API xap_client_fault : std::logic_error {
public:
    xap_client_fault() throw() : std::logic_error("") {}
    explicit xap_client_fault(const char* w) throw() :
```
std::logic_error(w) {}
virtual ~xap_client_fault() throw() {}
}

class XAP_API xap_error : std::runtime_error {
public:
    virtual ~xap_error() throw() {}
    virtual const char* getContext() const throw() {
        return (m_context.c_str());
    }
    virtual const int getLine() const throw() {
        return (m_line);
    }
protected:
    xap_error() throw() : std::runtime_error("") {} 
    explicit xap_error(const char *const w) throw() :
    std::runtime_error(w) {} 
    virtual void setContext(const char* c) {
        m_context = c;
    }
    virtual void setLine(const int line) {
        m_line = line;
    }
private:
    std::string m_context;
    int m_line;
};

class XAP_API xap_bad_feature : public xap_client_fault {
public:
    xap_bad_feature() throw() : xap_client_fault(XAP_FAULT_BAD_FEATURE) {}
};

class XAP_API xap_bad_type : public xap_client_fault {
public:
    xap_bad_type() throw() : xap_client_fault(XAP_FAULT_BAD_TYPE) {}
};

class XAP_API xap_bad_path : public xap_client_fault {
public:
    xap_bad_path() throw() : xap_client_fault(XAP_FAULT_BAD_PATH) {}
};

class XAP_API xap_bad_access : public xap_client_fault {
public:
    xap_bad_access() throw() : xap_client_fault(XAP_FAULT_BAD_ACCESS) {}
};
class XAP_API xap_bad_number : public xap_client_fault {
public:
    xap_bad_number() throw() : xap_client_fault(XAP_FAULT_BAD_NUMBER) {} 
};

class XAP_API xap_bad_xap : public xap_error {
public:
    xap_bad_xap() throw() : xap_error(XAP_ERR_BAD_XAP) {}
    explicit xap_bad_xap(const char *const c) :
        xap_error(XAP_ERR_BAD_XAP) {
            setContext(c);
            setLine(0);
        }
};

class XAP_API xap_bad_xml : public xap_error {
public:
    xap_bad_xml() throw() : xap_error(XAP_ERR_BAD_XML) {}
    xap_bad_xml(const char *const c, const int l) :
        xap_error(XAP_ERR_BAD_XML) {
            setContext(c);
            setLine(l);
        }
};

class XAP_API xap_no_match : public xap_error {
public:
    xap_no_match() throw() : xap_error(XAP_ERR_NO_MATCH) {} 
    explicit xap_no_match(const char *const path) :
        xap_error(XAP_ERR_NO_MATCH) {
            setContext(path);
            setLine(0);
        }
};
Runtime Flow of Control

This roadmap will follow the most important code paths through the code. Once you are familiar with these paths, you should be able to find your way around the less important highways and byways.

**MetaXAP::parse**

Until the last buffer is encountered, `XAPTk_Data::parse` is used, which does some pre-parsing to deal with end-of-line filtering. Once that is dealt with, the buffers are handed over to `XAPTk_Data::innerParse`, which does the actual filtering, and eventually calls `XAPTk::DOMGlue_Parse` (in DOMGlue.cpp). This is where the real parsing occurs. It passes through the DOM code to the underlying expat parser. A DOM tree gets built up as the XML is parsed (`XAPTk_Data::m_domDoc`). This DOM doc is an exact representation of the XML syntax that was parsed (modulo comments, XML processing instructions, parsed entities, etc., which are irrelevant for RDF).

After the last buffer is parsed, `XAPTk_Data::loadFromTree` is called, which is where the normalization is done. The objective is to convert the exact representation of the XML serialization into a representation that is easier to manipulate. This normalized representation, which folds the many-equivalent syntax representations into one model, is a forest of trees. Each tree is represented by a class `NormTree` object. Each tree has a non-descript root, and contains all the properties that are defined for a particular schema/namespace, or for a particular ID. The ID form has many uses, one of which is to manage the timestamps for properties. More on this later.

The class `RDFToNormTrees` normalizes the raw DOM tree into `NormTrees`. It is a gigantic DOMWalker (a pure virtual base class which implements depth-first, preorder tree walks). As the `RDFToNormTrees` walks the original DOM tree, it executes a finite state machine. This state machine has 6 states:

1) **state_init**
   - Looking for an `rdf:RDF` element.

2) **state_ignore**
   - Ignore this element (`m_beingIgnored`) and all of its children.

3) **state_rdf**
   - Found an `rdf:RDF` element, looking for an `rdf:Description` element.

4) **state_desc**
   - Found an `rdf:Description` (or `parseType='Resource'`, or implicit description), looking for properties.

5) **state_prop**

14 Sept 01
Found a property, looking for a value, a structure container, a nested description, or a special case (see code for details).

6) **state_container**

Found a container, looking for a list member.

A side-effect of certain state transitions is the construction of nodes in a **NormTree**. When the RDFToNormTrees object is finished walking the original tree, it deletes the original DOM Document, and leaves behind two std::map data structures XAPTk_Data::m_bySchema, and XAPTk_Data::m_byID. The former maps a schema/namespace name to a **NormTree** of RDF properties and values, the latter maps a schema/namespace name to a **NormTree** used to store timestamps (a stylized RDF bag of properties).

XAPTk_Data::loadFromTree continues by enumerating the schema/namespaces loaded in XAPTk_Data::m_bySchema. The corresponding **NormTree** in XAPTk_Data::m_byID is looked up by this namespace. The encoded timestamp properties are loaded into a more convenient data structure (XAPTk_ChangeLog, XAPTk_PunchCardByPath, and class PunchCard, all defined in XAPTk_Data.h). See XAPTk_Data::m_changeLog.

Finally, XAPTk_Data::loadFromTree returns. The last thing that **MetaXAP::parse** does is detects if aliasing is enabled. If so, it verifies that linked values that are defined are equal, and populates any linked values that were not defined. This is a side-effect of flipping the XAP_OPTION_ALIASING_ON flag, which calls VerifyAndPopulate (static module function in MetaXAP.cpp).

**MetaXAP::SetAlias**

After validating that the parameters are legal, an entry is added to the staticMetaXAP_aliasMap (defined in MetaXAP.cpp). This maps an alias property to an actual property.

Aliases are treated as linked values. This is implemented by actually instantiating all properties that share the same value, and setting/copying the value. This is done by VerifyAndPopulate (see above), and by each non-const function of MetaXAP that can alter property values, utilizing a pre-computed list of linked values generated by PreResolveAlias, which is called at the end of **MetaXAP::SetAlias**.

The job of PreResolveAlias is to resolve all alias lookups (and actual to alias reverse lookups), and build this information into a sparse matrix, implemented with nested std::map structures: MetaXAP_InfoMap and MetaXAP_ResolvedAliases, both defined in MetaXAP.cpp. The sparse matrix is stored in the static variable MetaXAP_resolvedAliases (in MetaXAP.cpp).
Both the alias and actual properties are entered into the MetaXAP_ResolvedAliases map as keys. The values are maps which list all of the other properties that are linked by value. So if I do this:

```cpp
MetaXAP::SetAlias("dc", "Foo", "xap", "Bar");
// Alias = <dc,Foo>
// Actual = <xap,Bar>
```

The MetaXAP_ResolvedAliases structure will contain:

```cpp
MetaXAP_ResolvedAliases: {
[<dc,Foo>] = MetaXAP_InfoMap : {
    [<xap,Bar>] = MetaXAP_AliasInfo: {
        actual = true;
        aliasSingle = true;
    }
}
[<xap,Bar>] = MetaXAP_InfoMap : {
    [ dc,Foo ] = MetaXAP_AliasInfo: {
        actual = false;
        aliasSingle = true;
    }
    [ xy,ZZY ] = MetaXAP_AliasInfo: {
        actual = false;
        aliasSingle = true;
    }
}
```

The meaning of aliasSingle, the four flavors of aliases, and the other fields of MetaXAP_AliasInfo are described in the comment above PreResolveAlias. Search for COMMENT_ALIAS_FLAVORS.

If another alias for <xap,Bar> is added, <xy,ZZY>, the structure will contain:

```cpp
MetaXAP_ResolvedAliases: {
[<dc,Foo>] = MetaXAP_InfoMap : {
    [<xap,Bar>] = MetaXAP_AliasInfo: {
        actual = true;
        aliasSingle = true;
    }
}
[<xap,Bar>] = MetaXAP_InfoMap : {
    [ dc,Foo ] = MetaXAP_AliasInfo: {
        actual = false;
        aliasSingle = true;
    }
    [ xy,ZZY ] = MetaXAP_AliasInfo: {
        actual = false;
        aliasSingle = true;
    }
}
```
Notice that the MetaXAP_InfoMap for <xap,Bar> now has two entries, which are the two properties whose values are linked to <xap,Bar>. This ensures that if the value for <xap,Bar> is changed directly, the other two properties will also get changed. More about how this works will be discussed in MetaXAP::set.

MetaXAP::get

Right away, XAPTk_Data::get is called. First, XPath is evaluated against the appropriate NormTree, looked up by schema/namespace name. If no node is found, it returns FALSE.

Next, the form is checked to make sure it is simple (you can’t do a get on anything but xap_simple).

If the node is an attribute, its value is returned.

If the node is an element, XAPTk_Data::extractPropVal is called, which in turn calls NormTree::getText. The children of the element are examined; if it has no children, an empty string is returned. If it has exactly one child that is a text node, its text value is returned. Otherwise, a number of special cases involving XAPFeatures have to be dealt with.

Notice that there were no aliases to deal with. That’s because the linked value implementation has already accounted for aliases. The value returned has already been copied from the actual by other code.

MetaXAP::set

After validating input parameters, any possible aliases, associated with this property via MetaXAP_CollectAliases, are collected.

BEGIN MetaXAP_CollectAliases

Remember, all non-const functions that alter property values call MetaXAP_CollectAliases, so this description also applies to append, remove, etc.

If aliasing is not enabled, don’t do anything.

Otherwise, the first objective is to find a valid value for the MetaXAP_ResolvedAliases::iterator entry. CheckAliases is used to see if this
path is an actual (target of aliases). If so (CheckAliases returns \textit{FALSE}), lookup the path in \texttt{MetaXAP\_resolvedAliases}, save if valid. Note that the conformed path is tried first (from \texttt{XAPTk::ConformPath}, in \texttt{xaptkfuncs.cpp}), which is the longest path prefix that contains no wildcards (*). Also, the structured container type (sct) is needed, which is normally filled in by CheckAliases, but since it returned \textit{FALSE}, it must be figured out. We get the \texttt{MetaXAP\_InfoMap}, and try to find a matching member. If not found, the full path (without conformance) is tried, since flavor 3 and flavor 4 (described in \texttt{COMMENT\_ALIAS\_FLAVORS}) have wildcards in their canonical actual paths. If not found, just use the first entry as a best guess. In any case, remember that the original path is an actual.

If CheckAliases returned true, we just get the matching entry, and \texttt{sct} is already assigned by CheckAliases.

If the pointer to the output parameter \texttt{cType} is not \texttt{NULL}, we assign \texttt{sct} to the variable it points at.

Our next objective is to massage the canonical path stored in the alias entry into an actual path that corresponds to the one passed into \texttt{MetaXAP\_CollectAliases}. The variable \texttt{savedPath} holds any variable part of the path that was detected during CheckAlias or ConformPath earlier. If it is non-empty, we need to remember to tack it on any container paths we collect as target linked values. If the original path was an alternate by language, remember that too. We need to determine if the target is single (not a container). \texttt{xap\_sct\_unknown} means single. If \texttt{sct} is some other value and the original path is not actual, \texttt{isSingle} is \texttt{TRUE} only if we are flavor 3 or 4.

If the \texttt{savedPath} has \texttt{last()} in the predicate, we convert it to the appropriate canonical path.

Now that we have all of the information we need, we build a list of target paths for linked values. We iterate over the \texttt{MetaXAP\_InfoMap} value of entry. The iterator is item. We do a little extra work to guarantee that the first slot (0) in the list is always the target of the actual, which is always in the second slot (1). This is easy when the original path was an alias (just put the original path is slot 0, and the looked up actual in slot 1). It’s harder when the original path was an actual, we have to pick some alias path to put into slot 0: that’s why there is a big block of code that starts “if (isActual)”. We arrange all this by saving the corresponding paths in \texttt{matchOrig}, \texttt{fullOrig}, \texttt{matchActual}, and \texttt{fullActual}.

We’re building our list in the output parameter \texttt{props}, which is a vector. Normally, we just want to put \texttt{fullOrig} in the first slot, and \texttt{fullActual} in the second. However, there is one special case where the original path was an actual, and has targets, but none of the targets qualify for one reason or another. For example, if the actual is member 2 of a
container, but all aliases are either targeted at member 1 or the whole container, nothing actually matches. See the comment in the block that starts:

```java
if (isActual && !(foundActual && foundAlias)) {
```

The items in MetaXAP_InfoMap are searched, skipping matches for actual and its alias, since they are already loaded in the list. If any fixup is needed, we append the variable parts as needed. Finally, we return `TRUE`.

**END MetaXAP_CollectAliases**

If there were no aliases collected, just call `XAPTk_Data::set`. Otherwise, loop through the list of linked values, and call `XAPTk::set` on each, catching and ignoring errors for all but the original path.

In `XAPTk_Data::set`, we evaluate the path and convert character escapes to raw characters. If we evaluate to a node, we replace its value with `XAPTk_Data::replaceProp`. Otherwise, if the container type is unknown (not a container), we call `XAPTk_Data::createProp`. If it is a container, we figure out what type. If the container does not exist, we call the type-specific form of `XAPTk_Data::createFirstItem`, otherwise, `XAPTk_Data::append` is called.

In `XAPTk_Data::replaceProp`, NormTree is looked up and a determination is made if this is an element or an attribute. The appropriate form of `NormTree::setText` is called, and also update the timestamp by calling `XAPTk_Data::punchClock`.

In `XAPTk_Data::setText`, handle special cases and features, then set the text child to the value passed in.

In `XAPTk_Data::createProp`, we lookup the NormTree, creating one if needed. We use the form of `NormTree::evalXPath` which creates a node if one is not found. The rest of the code looks just like `replaceProp`.

In `XAPTk_Data::createFirstItem`, we create the container of the appropriate type, and then create the first member item. The rest of the code is just like `replaceProp`, except that we set the timestamp on the entire container, rather than individual members.

In `XAPTk_Data::append`, we find the member item specified, climb the tree to get information about the container (parent), and then create a new node and place it as specified by the input parameters. We set the timestamp on the whole container.

**MetaXAP::enumerate**

All forms of enumerate directly call `XAPTk_Data::enumerate`.

In `XAPTk_Data::enumerate`, figure out if everything is being enumerated, or just certain schemas, subPaths, or depths in steps. For each schema, call `NormTree::enumerate`.

In `NormTree::enumerate`, we create a Paths object (`Paths.cpp`), and construct a `DW4 enumeratePropElem DOMWalker`, passing the Paths object as a parameter. `DW4 enumeratePropElem` is defined in `NormTree.cpp`. It basically walks the tree, and for
each element that meets the input criteria (number of steps, or leaf nodes only), it computes a canonical path and appends it to the Paths object.

**MetaXAP::serialize**

In `XAPTk_Data::serialize`, we first deal with header information, then the `rdf:RDF` boilerplate. Then we iterate through each of the normalized trees in `m_bySchema`. We call `NormTree::serialize` for each one. Then we tack on the timestamp info, if any, with `XAPTk_Data::serializeTimestamps`, then more boilerplate and trailer stuff.

In `NormTree::serialize`, which is implemented in `NormTree_serialize.cpp`, we arrange for the proper line ending, and then we add the boilerplate for `rdf:Description`, which is one top-level per schema. We loop through all the namespace definitions, and write out any that we need. Finally, we construct a DOMWalker to serialize the `NormTree`: a `SerializePretty` for pretty-printing, a `SerializeCompact` for compact notation.

Both DOMWalkers handle all the nasty details of writing out the syntax. There are many special cases to handle. See comments in the code for details. There’s also a big block of comments at the beginning of `NormTree.cpp`, which explains the internal layout of `NormTrees`.

In `XAPTk_Data::serializeTimestamps`, We iterate through `m_changeLog`. Each entry is a `XAPTk_PunchCardByPath` map, which contains a timestamp entry for each property that changed. The body of the loop writes out an `rdf:Description` with an ID that is set to the namespace name for each schema that has properties that were changed. There is one property, `XAPTK_TAG_TS_CHANGES`, which is a `Bag`. Each member item of the `bag` is a timestamp entry, written in a compact, comma separated value notation.

**NormTree::evalXPath**

This simple XPath evaluator uses a very restricted subset of the XPath notation (see Section 3.3.1, “XPath Syntax”). It takes the input expression and separates each step by parsing out the slashes with `XAPTk::ExplodePath` (defined in `xaptkfuncs.cpp`). For each step, we do a lexical analysis, and then an evaluation. The side-effects of the evaluation is a Node pointer, stored in current. XPaths always evaluate to a single Node, rather than a node list.

The lexical analysis generates `XAPTk_Token` class objects (defined in here), which are appended to a `VectOToken` (defined here). Begin and end iterators to the original step string are saved in the token for type `tChars`. All token types start with “t”.

In evaluation, we use a finite state machine, which may be described as follows:

1) **sInit**
   - On `tDot`, next state is `sEmpty`.
   - On `tAt`, next state is `sAttr`.
   - On `tStar`, next state is `sList`. 
On tChars, next state is sName: if there are no more tokens, recover the element node name and look it up with `NormTree::selectChild`. If not found but required, create a node. Set current an continue to next step.
Otherwise, throw `xap_bad_path`.

2) sEmpty
If there are no more tokens, set return value to current.
Otherwise, throw `xap_bad_path`.

3) sAttr
On tStar, if there are no more tokens, set return value to current, else throw `xap_bad_path`.
On tChars, cast current to `Element*`. If `NULL`, or there are more tokens, throw `xap_bad_path`. Otherwise, recover the attribute name from the token, get the attribute, create it if required, and set current to it.
Otherwise, throw `xap_bad_path`.

4) sList
On tLB, throw `xap_bad_paths` if boundary conditions not met, otherwise next state is sPred and save some state.
Otherwise, if there are more tokens, throw `xap_bad_path`, else set return value to `current`.

5) sName
On tLB, next state is sPred.
On tParens, recover function name from token. If name is not supported, throw `xap_bad_path`, else set return value to `NULL` since functions are not yet supported.
Otherwise throw `xap_bad_path`.

6) sPred
On tAt, next state is pAttr.
On tChars, if token is not a number, next state is pName, else it is pOrd. Remember the left hand side (lhs) by assigning the current token index (tix) to it.
Otherwise throw `xap_bad_path`.

7) pAttr
On tChars, next state is pAName, remember left hand side (lhs) by assigning current token index (tix) to it.
Otherwise throw `xap_bad_path`. 
8) **pAName**
   
   On tEquals, next state is pMatch.
   Otherwise throw xap_bad_path.

9) **pName**
   
   On tEquals, next state is pMatch.
   On tParens, next state is pFunc.
   Otherwise throw xap_bad_path.

10) **pFunc**
    
    On tRB, recover function name from token; if it isn’t “last”, throw xap_bad_path. Set current to the last child of former value of current.
    Otherwise throw xap_bad_path.

11) **pMatch**
    
    On tChars, set the right hand side (rhs) to the current token index, and next state is pVal.
    Otherwise throw xap_bad_path.

12) **pVal**
    
    On tRB AND this is the last token, perform the match, creating the node if required. Assign it to current.
    Otherwise throw xap_bad_path.

If at any point ret != NULL, and we are at the last token or step, break out of the loop.
### MetaXAP Static Functions (Class Methods)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MetaXAP::Clone</strong></td>
<td>Makes a deep-copy of the MetaXAP object and returns it.</td>
</tr>
<tr>
<td><strong>MetaXAP::EnumerateAliases</strong></td>
<td>Returns a pointer to an object that enumerates all of the aliases defined for all MetaXAP objects.</td>
</tr>
<tr>
<td><strong>MetaXAP::set</strong></td>
<td>Extracts an externally saved serialization and saves as a string in a specified buffer.</td>
</tr>
<tr>
<td><strong>MetaXAP::GetAlias</strong></td>
<td>Gets the alias for the specified path, if any.</td>
</tr>
<tr>
<td><strong>MetaXAP::Merge</strong></td>
<td>Creates a new MetaXAP object containing merged metadata.</td>
</tr>
<tr>
<td><strong>MetaXAP::RegisterNamespace</strong></td>
<td>Register a namespace name (should be a URI), and a suggested prefix for composing qualified names.</td>
</tr>
<tr>
<td><strong>MetaXAP::RemoveAlias</strong></td>
<td>Removes the specified alias from the alias map for all metadata objects.</td>
</tr>
<tr>
<td><strong>MetaXAP::SetAlias</strong></td>
<td>Adds to the alias map for all instances of MetaXAP.</td>
</tr>
</tbody>
</table>

### MetaXAP Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MetaXAP::XAPClock</strong></td>
<td>Clients provide the clock used for creating timestamps.</td>
</tr>
<tr>
<td><strong>MetaXAP::XAPChangeBits</strong></td>
<td>Each timestamp record includes an indication of how the property was last changed.</td>
</tr>
</tbody>
</table>

### MetaXAP Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public default constructor</td>
<td>Create an empty object with no clock.</td>
</tr>
<tr>
<td><strong>MetaXAP ()</strong></td>
<td></td>
</tr>
<tr>
<td>public construct empty with clock</td>
<td>Creates an empty object with a clock.</td>
</tr>
<tr>
<td><em><em>MetaXAP (XAPClock</em> clock)</em>*</td>
<td></td>
</tr>
</tbody>
</table>
### MetaXAP Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public construct from buffer</code> MetaXAP</td>
<td>Constructs a populated MetaXAP from a single buffer of raw XML.</td>
</tr>
<tr>
<td><code>~MetaXAP()</code></td>
<td>Destroy this object and all internally allocated memory.</td>
</tr>
<tr>
<td><code>MetaXAP::append</code></td>
<td>Creates a new property with the specified <code>value</code>, and add it after the property specified by <code>namespace ns</code> and <code>path</code>.</td>
</tr>
<tr>
<td><code>MetaXAP::count</code></td>
<td>Returns the number of items in the structured container specified by <code>ns</code> and <code>path</code>.</td>
</tr>
<tr>
<td><code>MetaXAP::createFirstItem</code></td>
<td>Creates a structured container of the specified type.</td>
</tr>
<tr>
<td><code>MetaXAP::enable</code></td>
<td>Enables or disables the specified option(s), such as <code>XAP_OPTION_DEBUG</code>.</td>
</tr>
<tr>
<td><code>MetaXAP::enumerate</code></td>
<td>Enumerates MetaXAP object properties.</td>
</tr>
<tr>
<td><code>MetaXAP::extractSerialization</code></td>
<td>Extracts an externally saved serialization and saves as a string in a specified buffer.</td>
</tr>
<tr>
<td><code>MetaXAP::get</code></td>
<td>Gets the value at the property specified by <code>ns</code> and <code>path</code> as a string.</td>
</tr>
<tr>
<td><code>MetaXAP::getContainerType</code></td>
<td>Returns the type of the specified container.</td>
</tr>
<tr>
<td><code>MetaXAP::getForm</code></td>
<td>Returns the type of property specified by <code>ns</code> and <code>path</code>.</td>
</tr>
<tr>
<td><code>MetaXAP::getResourceRef</code></td>
<td>Returns the reference (URI) for the resource that this MetaXAP is about.</td>
</tr>
<tr>
<td><code>MetaXAP::getTimestamp</code></td>
<td>Returns <code>FALSE</code> if the property specified by <code>ns</code> and <code>path</code> is not defined. Otherwise, returns <code>true</code>.</td>
</tr>
<tr>
<td><code>MetaXAP::isEnabled</code></td>
<td>Returns whether the specified option is enabled, such as <code>XAP_OPTION_DEBUG</code>.</td>
</tr>
<tr>
<td><code>MetaXAP::parse</code></td>
<td>Parses a buffer of XML and create the corresponding XMP objects.</td>
</tr>
<tr>
<td><code>MetaXAP::purgeTimestamps</code></td>
<td>Purges all timestamp records for properties with any <code>XAPChangeBits</code> set in <code>how</code>.</td>
</tr>
<tr>
<td><code>MetaXAP::remove</code></td>
<td>Removes the specified property and all of its sub-properties, if any.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>MetaXAP::serialize</code></td>
<td>Serializes the MetaXAP tree as XML.</td>
</tr>
<tr>
<td><code>MetaXAP::set</code></td>
<td>Sets the specified value at the end of the specified path, with the optionally specified features.</td>
</tr>
<tr>
<td><code>MetaXAP::setTimestamp</code></td>
<td>Sets the timestamp.</td>
</tr>
<tr>
<td><code>MetaXAP::setup</code></td>
<td>Enables client application to provide metadata to this instance of MetaXAP for automatic tracking.</td>
</tr>
<tr>
<td><code>MetaXAP::setResourceRef</code></td>
<td>Sets the reference to the resource (URI) that this MetaXAP is about.</td>
</tr>
</tbody>
</table>

**UtilityXAP Static Functions (Class Methods)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>UtilityXAP::cAnalyzeStep</code></td>
<td>Removes last step from path, break it into pieces.</td>
</tr>
<tr>
<td><code>UtilityXAP::CompareTimestamps</code></td>
<td>Compares the property with the specified namespace and path in instance a with instance b.</td>
</tr>
<tr>
<td><code>UtilityXAP::CreateXMLPacket</code></td>
<td>Use this routine to compute the header and trailer string for a packet, which you use yourself to create a XMP packet.</td>
</tr>
<tr>
<td><code>UtilityXAP::FilterPropPath</code></td>
<td>Filters UI text into valid XPath.</td>
</tr>
<tr>
<td><code>UtilityXAP::GetBoolean</code></td>
<td>Gets a property value as a boolean as specified by ns and path.</td>
</tr>
<tr>
<td><code>UtilityXAP::GetDateTime</code></td>
<td>Gets a property value as a date and time.</td>
</tr>
<tr>
<td><code>UtilityXAP::GetInteger</code></td>
<td>Gets a property value as an integer.</td>
</tr>
<tr>
<td><code>UtilityXAP::GetReal</code></td>
<td>Gets a property value as a real.</td>
</tr>
<tr>
<td><code>UtilityXAP::IsAltByLang</code></td>
<td>Returns <code>TRUE</code> if the specified path evaluates to a member of a structured container that is of type <code>xap_alt</code>, and which is selected by the attribute <code>xml:lang</code>.</td>
</tr>
<tr>
<td><code>UtilityXAP::SetBoolean</code></td>
<td>Sets a property value as a boolean.</td>
</tr>
<tr>
<td><code>UtilityXAP::SetDateTime</code></td>
<td>Sets the property value as a date and time.</td>
</tr>
<tr>
<td><code>UtilityXAP::SetInteger</code></td>
<td>Sets property value as an integer.</td>
</tr>
<tr>
<td><code>UtilityXAP::SetLocalizedText</code></td>
<td>Sets the structured container language alternation property.</td>
</tr>
<tr>
<td><code>UtilityXAP::SetReal</code></td>
<td>Sets a property value as a real.</td>
</tr>
</tbody>
</table>