xmp™

Adding Intelligence to Media

XMP TOOLKIT SDK
PROGRAMMER'S GUIDE

Adobe®

Contents

Preface

The [Extensible Metadata Platform (XMP)](#) provides a standard format for the creation, processing, and interchange of metadata, for a wide variety of applications.

About this document

This document, the *XMP Toolkit SDK Programmer's Guide*, provides guidance for programmers wishing to work with XMP metadata using the XMP Toolkit SDK, which is provided by Adobe® as a free Software Development Kit (SDK). This guide emphasizes tutorial material. The complete API reference is provided as separate HTML documentation in the SDK.

The public XMP Toolkit SDK contains two components, *XMPCore* and *XMPFiles*. The current public SDK, version 5.1.1, has C++ implementations of both and a Java implementation of XMPCore.

JAVASCRIPT: A JavaScript API for both components is available through the scripting interface of applications such as Adobe Bridge. The JavaScript API, known as XMPScript, is documented in the *JavaScript Tools Guide*. This document is available as part of the ExtendScript SDK, available at [Adobe Developer's Center (Scripting)](#), and also with the Adobe Bridge SDK, available at [Adobe Developer's Center (Adobe Bridge)](#).

How this document is organized

This document has the following sections:

▶ [Chapter 1, "XMP Toolkit SDK Overview](#)," provides an introduction to the XMP Toolkit SDK components and usage.

▶ [Chapter 2, "The XMPCore Component](#)," describes the core API, which allows you to manipulate XMP metadata programmatically.

▶ [Chapter 3, "The XMPFiles Component](#)," describes the file-handler API, which allows you to read and write XMP metadata that is embedded in files.

▶ [Chapter 4, "Using the XMP Toolkit SDK](#)," explains how to get started using the XMP Toolkit SDK, and provides hands-on examples of how to perform typical metadata-handling tasks.

Conventions used in this document

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

Typeface Style	Used for:
`Monospaced bold`	XMP property names. For example, `xmp:CreateDate`
`Monospaced Regular`	XML code and other literal values, such as value types and names in other languages or formats

1   XMP Toolkit SDK Overview

The XMP Toolkit Software Development Kit (SDK) is available for download from the Adobe Developer Center (XMP).

The XMP Toolkit SDK provides an API for handling XMP metadata. This document supplies an overview of the API, together with hands-on example code and usage guidance. This is a companion document to the API reference documentation, also included in the XMP Toolkit SDK.

The XMP Data Model

The *XMP Specification*, available from Adobe Developer Center (XMP), provides a complete formal specification for XMP. Before working with the XMP Toolkit SDK, you must be familiar with, at a minimum, the XMP Data Model.

The specification has three parts:

▶ *Part 1, Data and Serialization Models* covers the basic metadata representation model that is the foundation of the XMP standard format. The Data Model prescribes how XMP metadata can be organized; it is independent of file format or specific usage. The Serialization Model prescribes how the Data Model is represented in XML, specifically RDF.

   This document provides all the details a programmer would need to implement a metadata manipulation system such as the XMP Toolkit SDK (which is available from Adobe).

▶ *Part 2, Standard Schemas*, provides detailed property lists and descriptions for standard XMP metadata schemas; these include general-purpose schemas such as Dublin Core, and special-purpose schemas for Adobe applications such as Photoshop®. It also provides information on extending existing schemas and creating new schemas.

▶ *Part 3, Storage in Files*, provides information about how serialized XMP metadata is packaged into XMP Packets and embedded in different file formats. It includes information about how XMP relates to and incorporates other metadata formats, and how to reconcile values that are represented in multiple metadata formats.

About the XMP Toolkit SDK

The latest version of the XMP Toolkit SDK is available from:

http://www.adobe.com/devnet/xmp/

The extracted archive places all of the SDK contents beneath a root folder named `XMP-Toolkit-SDK-5.1.1` (referred to in this document as *`<xmpsdk>`*).

The root folder contains these subfolders:

`/`	At the root level:
	`BSD_License.txt` The license agreement. `XMP-Toolkit-SDK-Overview.pdf` An overview document.
`build/`	Projects for building the C++ version of the XMP Toolkit SDK in Mac OS, Windows, and UNIX/Linux.
`docs/`	The three-part *XMP Specification*, this document, and the API reference documentation (`API/index.html`).
`java/`	The Java implementation of the XMPCore library, the Javadoc documentation, project files for Eclipse 3.x, and sample code.
`public/include/`	The header files used by clients of the XMP Toolkit SDK.
`samples/`	Sample and tutorial projects and the necessary resources to run the sample code.
`source/`	The source code that implements the XMP Toolkit SDK.

`third-party/`
  `expat/`
  `zlib/`
  `MD5/`
	Place holders for third party source files which are needed for the XMP Toolkit SDK, including `ReadMe.txt` files with information on how to obtain and install the tools. MD5 source code, needed by both components for MD5 hash computation, is included.

SDK components

The XMP Toolkit SDK contains two libraries, XMPCore and XMPFiles.

▶ Both XMPCore and XMPFiles are provided as C++ implementations with project files for:

▷ Windows XP and above (32-bit and 64-bit), using Visual Studio 2008 (VC++ 9)

▷ Mac OS X 10.4 and above (32-bit and 64-bit), using Xcode 3.x, creating binaries for Intel processors.

▷ Linux using makefiles for the GNU C Compile (`gcc`) version 4.2. Instructions for compiling under Linux are in `<xmpsdk>/build/gcc4/usage_examples.txt`.

▶ A Java implementation of XMPCore is also provided, to be used with J2SE Version 1.5 or higher. Project files for Eclipse 3.2 or higher and an Ant build file are included.

Dependencies

These publicly available components are also required to build the C/C++ libraries:

▶ Expat XML parser: You must provide the Expat XML parser 2.0 or higher. See
http://sourceforge.net/projects/expat/.

▶ ZLIB: Required by XMPFiles to work with compressed formats such as UCF. This public compression library is available from http://www.zlib.net.

The XMP libraries

The XMP Toolkit SDK contains two libraries, XMPCore and XMPFiles.

▶   XMPCore provides the fundamental API for manipulating, serializing, and deserializing XMP data, along with support utilities for building particular structures and for iteration. It does not deal with files. It is available in both C++ and Java implementations.

This document assumes the use of C++. Javadoc is available in the `java/XMPCore/docs` folder for the Java implementation of XMPCore.

▶   XMPFiles provides an API for convenient I/O access to the main, or document level, XMP for a file. It allows you to locate and retrieve existing metadata from a file, update file metadata, and add new metadata to a file. It contains file handlers for individual file formats, and a packet scanner that can be used for unknown file formats.

Typically, you will use XMPFiles to read XMP metadata from a file into an XMP object (represented by the concrete class `SXMPMeta` in the XMPCore component). You will use XMPCore functions to examine and manipulate the metadata, then use XMPFiles again to write it back into the file.

The following chapters provide overviews and API summaries for the components. This section discusses issues relevant to the Toolkit as a whole.

Template classes and accessing the API

The full client API is defined and documented in the `TXMP*.hpp` header files. The `TXMP*` classes are C++ template classes that must be instantiated with a string class such as `std::string`, which is used to return text strings for property values, serialized XMP, and so on.

To allow your code to access the entire XMP API you must:

▶   Provide a string class such as `std::string` to instantiate the template classes.

▶   Provide access to XMPCore and XMPFiles by including the necessary defines and headers.

To do this, add the necessary define and includes directives to your source code so that all necessary code is incorporated into the build:

```
#include <string>
#define XMP_INCLUDE_XMPFILES 1 //if using XMPFiles
#define TXMP_STRING_TYPE std::string
#include "XMP.hpp"
```

The SDK provides complete reference documentation for the template classes, but the templates must be instantiated for use. You can read the header files (`TXMPMeta.hpp` and so on) for information, but do not include them directly in your code. There is one overall header file, `XMP.hpp`, which is the only one that C++ clients should include using the `#include` directive. Read the instructions in this file for instantiating the template classes.

When you have done this, the API is available through the concrete classes named `SXMP*`; that is, `SXMPMeta`, `SXMPUtils`, `SXMPIterator`, and `SXMPFiles`. This document refers to the `SXMP*` classes, which you can instantiate and which provide static functions.

▶   Clients must compile `XMP.incl_cpp` to ensure that all client-side glue code is generated. Do this by including it in exactly one of your source files.

▶ Read `XMP_Const.h` for detailed information about types and constants for namespace URIs and option flags.

Multi-threading in the API

The functions in XMPCore and XMPFiles are thread safe. You must call the initialization and termination functions in a single-threaded manner; between those calls, you can use threads freely, following a multi-read, single-writer locking model. All locking is automatic and transparent.

After initialization, several readers can access an object concurrently, but a writer locks the object for every thread until the operation is complete. The locking occurs automatically when a write operation is attempted. For example, if you call `SetProperty()`, all other incoming read and write requests are queued until that operation iscomplete and the lock released.

The SDK allows you to configure and change the XMPCore and XMPFiles locking mechanism through conditional compilation. For example, you can choose to use a global lock instead of the multi-read/single writer model. You can choose to use native Mac OS or Windows APIs, or even Boost for the lock implementation. For details, see the file `XMP_LibUtils.hpp` in `<sdkroot>/source/`.

When using the toolkit in a multi-threaded environment, you should be aware that it is the hidden internal library object that is locked, not the client object. Multiple client objects can refer to the same internal object. For example, if you call `SetProperty()` on one client object, that change appears in all `GetProperty()` calls on all other client objects that refer to the same internal object.

Multiple references to the same library object are created by assignments (a = b;) which perform a shallow copy of the client object. The `Clone()` method creates a deep copy of the internal library object; see "Copying metadata" on page 34.

Error handling

All of the API functions throw exceptions for serious errors. The `XMP_Error` exception object includes a numeric code and diagnostic string; use `XMP_Error::GetId()` and `XMPError::GetErrorMsg()` to retrieve them.

New numeric codes may be added at any time. There are typically many possible explanations for each numeric code; the error explanations try to be precise about the specific circumstances causing the error. The error explanation strings are intended for debugging use only; they are not localized, and should not be displayed to users.

Data-dump utilities also provide debugging assistance; see "Examining XMP objects" on page 17.

Sample code and tools

The SDK provides a set of samples that illustrate coding techniques for various tasks. In addition to the source code for each sample, there is a project file for use with a platform-specific IDE. All samples offer both 32-bit and 64-bit targets.

▶ Project files for MS Visual Studio 2008 are in the folder `<xmpsdk>\samples\build\vc9`.

▶ Project files for XCode 3.x are in the folder `<xmpsdk>/samples/build/xcode3`

▶ Build files for GCC are in `<xmpsdk>/samples/build/gcc4`

The source code for the samples is in *<xmpsdk>*/samples/source. When you build them, the compiled code is written to *<xmpsdk>*/samples/target/, to a platform-specific folder with debug and release subfolders.

These sample console applications are provided:

ReadingXMP *filename*	Demonstrates the basic use of the XMPFiles and XMPCore components, obtaining read-only XMP from a file and examining it through the XMP object.
	Takes one parameter, the file name. Prints results to the screen, and to an output file named XMPDump.txt.
	For a tutorial walkthrough based on this sample, see <u>"Walkthrough 1: Opening files and reading XMP" on page 58</u>.
ModifyingXMP *filename*	Demonstrates how to open a file for update, and modifying the contained XMP before writing it back to the file.
	Takes one parameter, the file name. Prints results to the screen, and to two output files named XMP_RDF.txt and XMP_RDF_compact.txt
	For a tutorial walkthrough based on this sample, see <u>"Walkthrough 2: Modifying XMP" on page 69</u>.
CustomSchema	Demonstrates how to work with a custom schema that has complex properties. It shows how to access and modify properties with complex paths using the path composition utilities from the XMP API.
	Takes no parameters.
	Output files are named CS_RDF.txt,NameDump.txt, and XMPDump.txt.
	For a tutorial walkthrough based on this sample, see <u>"Walkthrough 3: Working with a custom schema" on page 73</u>.
XMPCoreCoverage XMPFilesCoverage	These demonstrate syntax and usage by exercising most of the API functions of each XMP Toolkit SDK component, using a sample XMP Packet that contains all of the different property and attribute types.
	These commands take no parameters.
	Output files are named XMPCoreCoverageLog.txt, and XMPFileCoverageLog.txt.
XMPIterations	Demonstrates how to use the iteration utility in the XMPCore component to walk through property trees.
	Takes no parameters; the file name "../../../testfiles/Image1.jpg" is specified in the code.
DumpMainXMP *filename*	Uses the XMPFiles component API to find the main XMP Packet for a data file, serialize the XMP, and display it.
	Takes one parameter, the file name.

`DumpScannedXMP` *filename*	Scans a data file to find all embedded XMP Packets, without using the XMPFiles API or smart handlers. If a packet is found, serializes the XMP and displays it. This is not a particularly efficient scan, and is meant for use in debugging.
	Takes one parameter, the file name.

In addition, these command-line tools are provided:

dumpfile

```
dumpfile [-help | -version | [-tree|-keys|-list] | -nocomments] path ]
```

Recursively parses the structure of the given file and prints a view of the file structure to standard output. The tool identifies chunks, subchunks, fields, and properties (in a broad sense), and creates a tree structure where each node contains a triple of <*key*, *value*, *comment*>; the *comment* portion is anything that is not a value, but can still be useful for understanding or validating the content. If an XMP Packet is found, it is identified.

This tool is not intended to extract metadata from files, or for any kind of use in production. It is a development aid that can help you determine whether a file that cannot be read by XMPFiles is malformed.

Switches

`-help`	Prints usage information for the command.
`-version`	Prints version information for this tool, and for the version of XMPCore to which it is statically linked. This tool does not use XMPFiles.
`-tree`	Shows the tree structure of the file. This is the default.
`-keys`	Shows only the key-value nodes found in the file, as a list with no hierarchical structure, in alphabetical order by key.
`-list`	Shows only the key-value nodes found in the file, as a list with no hierarchical structure, in the order of parsing.
`-nocomments`	Does not show the comment portion of key-value nodes.

xmpcommand

```
xmpcommand [-help | -out | -safe | -smart | -scan | -compact]
    [info mediafile | put xmpfile mediafile | get mediafile |
     dump mediafile]
```

A command-line tool for performing basic XMP actions such as get, put, and dump. Can be used for testing and scripting automation.

Returns 0 if there are no errors. Warnings and errors are printed as part of the output, either to `stdout` or the output file, not to `stderr`; you should check the return value before using the output.

Switches

`-help`	Prints usage information for the command, with examples.

`-out outfile`	Writes output and logs all warnings and errors *both* to standard output *and* to the specified output file. If you specify the output file without this switch, `stdout` is not used.
`-safe`	Updates safely, writing to a temporary file then renaming it to the original file name. See API documentation for `safeSave(kXMPFiles_UpdateSafely)`.
`-smart`	Requires the use of a smart file-format handler, does no packet scanning. Use of smart handlers is the default, if one is available.
`-scan`	Forces packet scanning, does not use a smart file-format handler.
`-compact`	Writes extracted XMP Packet in compact RDF-style, rather than pretty-printing attribute value for readability (which is the default).

Actions

`info mediafile`	Prints basic information about the file.
`put xmpfile mediafile`	Injects the XMP contained in the specified XMP file into the specified media file.
`get mediafile`	Retrieves the XMP Packet contained in the specified media file.
`dump mediafile`	Prints the XMP Packet contained in the specified media file to standard output. Preferred to `get` for testing output.

EXAMPLES:

```
xmpcommand info Sample.jpg
xmpcommand get ../Sample.jpg  >onlyFileOut.txt
xmpcommand -out alsoFileOut.txt get Sample.eps
xmpcommand put xmp_mySnippet.xmp Sample.jpg
xmpcommand -smart put xmp_mySnippet.xmp Sample.jpg,exename
```

(Note that in the last example, a smart handler would be used by default for a JPEG file, or any other supported file type, even without the `-smart` switch.)

2   The XMPCore Component

The XMPCore component of the XMP Toolkit SDK provides the fundamental API for manipulating and serializing XMP data, along with support utilities for building particular structures and for iteration. It does not deal with files.

This chapter introduces the XMPCore component and describes how it relates to the XMP Data Model.

▶ "Reading XMP properties" on page 13 provides a brief overview of the XMP Data Model and describes the basic property-access and debugging capabilities of the XMPCore component.

▶ "Modifying XMP data in the XMP object" on page 19 discusses how to use XMPCore to modify XMP by creating and deleting properties and modifying property values.

▶ "Working with schemas" on page 27 discusses how to create a new schema with a unique namespace, and how to extend existing schemas by creating new properties.

▶ "Iterating over metadata" on page 29 discusses how to use the iteration utility in XMPCore to traverse a metadata tree in an XMP object.

▶ "API summary: the XMPCore component" on page 32 summarizes the functions provided by the main XMPCore classes.

The XMPCore component is available in both C++ and Java implementations. This guide discusses the C++ API, but the usage model for the Java implementation is similar.

For reference details of the C++ API, see the HTML documentation of the template classes, provided in the SDK under *<xmpsdk>*/docs/API. Java documentation is under `java/XMPCore/docs`.

Reading XMP properties

NOTE: This provides only a brief overview of the XMP Data Model; before working with the XMP Toolkit SDK, developers should understand the XMP Data Model, as documented in the *XMP Specification Part 1, Data and Serialization Models*.

XMP properties are simple or complex:

▶ Simple properties have literal values such as strings and Booleans

▶ Arrays and structures are sets of related values.

    ▷ Arrays are sets of indexed *items*, with each item holding a value.

    ▷ Structures are sets of named properties (*fields*), with each field holding a value.

Structures and arrays can contain other structures or arrays, nested to any depth. See the XMP Specification for complete details on data types and properties.

In addition to these basic types, there is special handling for Property qualifiers and language alternatives, and for Dates and times.

Basic property types

The `SXMPMeta` class provides a basic property accessor, `GetProperty()`, which is completely general, and additional accessors that are specialized for specifying items in arrays and structures. Call these functions in an instance of the `SXMPMeta` concrete class.

Because arrays and structures can be nested, the path to a particular item can be quite complex; the `SXMPUtils` class provides utilities for constructing paths, which you can then pass to the accessors. These helper functions are all static; call them directly from the `SXMPUtils` concrete class. There is no need to create `SXMPUtils` objects.

Simple properties

The easiest way to access simple properties is with the function `SXMPMeta::GetProperty()`. Provide the function with the namespace URI, property name and a string pointer to store the value. You can also pass a pointer to an `XMP_OptionBits` structure in which to return option flags that describe the property; if you do not want that information, the pointer can be null.

```
meta.GetProperty( kXMP_NS_XMP, "CreatorTool", &value, &opts );
```

The property name can be a general path expression to a property located somewhere other than at the top level.

The `GetProperty()` function returns true if the desired property exists and the reference variable are set accordingly. When the function returns false, indicating that the property does not exist, the reference variables (*&value* and *&option*) are not guaranteed to contain any meaningful values. You should always check the function's return value to discover whether you can depend on the contents of the reference variables:

```
bool result = meta.GetProperty( kXMP_NS_XMP, "CreatorTool", &value, &options );
if( result == false )
    value.clear();
```

Arrays and structures

To access the elements of a top-level array, use `SXMPMeta::GetArrayItem()`. The function signature is similar to that of `GetProperty()` except that you must pass the index of the desired element:

```
meta.GetArrayItem( kXMP_NS_DC, "creator", 1, &value, &opts );
```

This is a 1-based index; that is, the index for the first element is 1, not 0.

You can use `CountArrayItems()` to discover the number of elements in the array. Pass the namespace URI for the array and the array name:

```
int numItems = meta.CountArrayItems( kXMP_NS_DC, "subject" );
```

A special index value, the constant `kXMP_ArrayLastItem`, allows you to access the last element, regardless of how many elements are present.

Structures are analogous to associative arrays as they have named fields. Structures can have their own namespaces; to access a structure's field you must supply the structure's namespace URI and the field

name. Field namespaces (unlike schema namespaces) cannot be referenced using prefixes. The URI and field name are passed separately:

```
... fieldNamespaceURI, "MyFieldName"...
```

Because arrays and structures can contain nested arrays and structures, you may need a *path* to access an item or field value below the top level. In the XMP Toolkit SDK, paths are similar to, but not identical to, those defined by the XML path language, XPath. It is highly recommended that you use the provided utility functions to construct complex paths, rather than constructing them by hand.

Use the `Compose...Path()` methods in the `SXMPUtils` class to compose paths. These are static methods, called on class itself. For example, to compose the path to a field in a structure:

```
SXMPUtils::ComposeStructFieldPath( structNamepace, "MyStruct",
    fieldNamespace, "MyFieldName", &path );
```

This composes a path to the field 'MyFieldName' within the 'MyStruct' structure, and stores the composed path in the *path* variable. You can then use this composed path to get the property value with `GetProperty()`.

The composition utilities are very useful for complex, multi-step cases with nested structures or arrays, or for use with the retrieve-by-type accessor functions such as `GetProperty_Int()`. For example the following retrieves a Boolean value from the 'Flash' structure to determine whether the flash was fired:

```
bool value;
string path;

//compose the path
SXMPUtils::ComposeStructFieldPath( kXMP_NS_EXIF, "Flash", kXMP_NS_EXIF,
    "Fired", &path );

//pass the composed path to the accessor
meta.GetProperty_Bool( kXMP_NS_EXIF, path.c_str(), &value, NULL );
```

For additional detail, see <u>"Composing paths to complex properties" on page 22</u>.

Special value handling

The API provides helper functions for dealing with more complex values, including language alternatives and date-times.

Property qualifiers and language alternatives

Properties themselves may have their own properties attached to them. These `"properties of properties"` are known as property *qualifiers*. Language alternatives, called *alt-text items*, are a special case of qualified properties, indicated by the qualifier `xml:lang`. This is an example of a qualifier identified by a name and a namespace prefix. To retrieve the value for a property qualifier, you specify the qualifier namespace URI and the qualifier name:

```
meta::GetQualifier( schemaNamespaceURI, propertyName, qualNS, qualName,
    &qualVal, opts )
```

A language alternative array allows a text property to be chosen based on a desired language, so that the property value can be localized for several different languages. Each item in the array has a property qualifier named `xml:lang`. The value of the `xml:lang` qualifier is used to determine which language should be selected.

The easiest way to access alt-text items is with the `SXMPMeta::GetLocalizedText()` function, which has this signature:

```
bool SXMPMeta::GetLocalizedText (
    XMP_StringPtr schemaNS,
    XMP_StringPtr altTextName
    XMP_StringPtr genericLang,
    XMP_StringPtr specificLang,
    tStringObj * actualLang,
    tStringObj * itemValue,
    XMP_OptionBits * options )
```

The *genericLang* and *specificLang* parameters determine how an array item is selected. Both parameters are RFC 3066 language tags; see http://www.w3.org/International/articles/language-tags. The specific language tag is required and is used first to try and locate the desired element within the alt-text array. The optional generic language tag is used if there is no match for a specific language tag.

The specific language tag should consist of both primary and secondary subtags, for example `"en-US"`. (Case is not considered, but the secondary subtag is uppercase by convention.) The generic language tag is simply the first part of the specific language tag. For example if the specific language tag is 'en-US' then the generic language tag is 'en'. Although the specific tag can be supplied with just a primary tag it is recommended that the specific language tag has both primary and secondary subtags, for example:

```
// Providing a generic language of 'en' and a specific language of 'en-US'
meta.GetLocalizedText( kXMP_NS_DC, "title", "en", "en-US", NULL, &itemValue, NULL );
```

Ideally both the generic and specific tags should be supplied to `GetLocalizedText()`, as this gives the greatest chance of success of finding a suitable item.

The *actualLang* output parameter is used to store the actual language of the item that was selected, if any. The actual language is the value of the qualifier attached to the array item, for example 'en-US'.

An alt-text array can have a default item, which has the property qualifier value of 'x-default' and is known as the 'x-default' item. If present, it is always the first item of the array. The following shows a debugging dump of an `alt-text` property:

```
dc: http://purl.org/dc/elements/1.1 (0x80000000 : schema)
    dc:title (0x1E00 : isLangAlt isAlt isOrdered isArray)
        [1] = "An English Title" (0x50 : hasLang hasQual)
          ? xml:lang = "x-default" (0x20 : isQual)
        [2] = "An English Title" (0x50 : hasLang hasQual)
          ? xml:lang = "en-US" (0x20 : isQual)
        [2] = "Un Titre Francais" (0x50 : hasLang hasQual)
          ? xml:lang = "fr-FR" (0x20 : isQual)
```

You can see that the x-default qualifier is attached to the first element within the array. When `GetLocalizedText()` cannot match the specific language or the generic language, it returns the x-default item if there is one. If no item can be found, the function returns false.

To summarize, `GetLocalizedText()` takes the following steps:

1. Attempt to find and return an item with a qualifier that is an exact match of the specific language.

2. If no match is found, attempt to find and return a partial match from the generic language.

3. If no match is found, locate and return the item with the 'x-default' qualifier.

4. If no x-default item is present, return false and do not modify the output parameter.

For additional detail, see .

Dates and times

The `XMP_DateTime` class represents a date-time value. It can store times to nanoseconds, and provides fields to handle UTC times and time-zone offsets. Utility functions help you to handle and manage dates and times.

To retrieve a date value from metadata, use the accessor function `SXMPMeta::GetProperty_Date()`. This has a similar signature to the generic accessor function `GetProperty()` except that you pass an explicit `XMP_DateTime` pointer in which to store the retrieved date and time. If the property being accessed does not exist, the function returns false. For example, to access the date-time value of a property named 'MetadataDate':

```
XMP_DateTime myDate;
meta.GetProperty_Date( kMP_NS_XMP, "MetadataDate", &myDate, NULL );
```

For additional detail, see .

Examining XMP objects

If you wish to view XMP data you can produce a data dump of an XMP object. This allows you to view all properties of the XMP object. Examining XMP objects is a good way to become familiar with the XMP Data Model, as well as aiding in debugging.

To use `SXMPMeta::DumpObject()` you provide a callback function, which can output the contents of the XMP object to the console or to a file. For example, the following shows some standard properties in the Basic XMP schema:

```
xmp: http://ns.adobe.com/xap/1.0/ (0x80000000 : schema)
    xmp:ModifyDate = "2007-07-16T11:33:20+01:00"
    xmp:CreatorTool = "Adobe Photoshop CS3 Windows"
    xmp:CreateDate = "2007-07-16T11:33:20+01:00"
    xmp:MetadataDate = "2007-07-24T17:05:26+01:00"

dc: http://purl.org/dc/elements/1.1 (0x80000000 : schema)
    dc:title (0x1E00 : isLangAlt isAlt isOrdered isArray)
        [1] = "Updated English US" (0x50 : hasLang hasQual)
            ? xml:lang = "x-default" (0x20 : isQual)
        [2] = "Updated English US" (0x50 : hasLang hasQual)
            ? xml:lang = "en-US" (0x20 : isQual)
        [2] = "Updated English UK" (0x50 : hasLang hasQual)
            ? xml:lang = "en-GB" (0x20 : isQual)

    dc:description (0x1E00 : isLangAlt isAlt isOrdered isArray)
        [1] = "Green Bush" (0x50 : hasLang hasQual)
            ? xml:lang = "x-default" (0x20 : isQual)

    dc:subject (0x200 : isArray)
        [1] = "XMP"
        [2] = "SDK"
        [3] = "Test"
        [4] = "File"

    dc:format = "image/jpeg"
    dc:creator (0x600 : isOrdered isArray)
        [1] = "Author Name"
```

The dump shows all schemas, prefixes and properties of an object, and the option bits that describe each property. You can compare this format with the RDF for the same XMP, shown in "Parsing XMP" on page 53. The following extract highlights the different data components in the data dump:

```
dc: http://purl.org/dc/elements/1.1 (0x80000000 : schema)

    dc:description (0x1E00 : isLangAlt isAlt isOrdered isArray)
        [1] = "Green Bush" (0x50 : hasLang hasQual)
            ? xml:lang = "x-default" (0x20 : isQual)

    dc:subject (0x200 : isArray)
        [1] = "XMP"
        [2] = "SDK"
        [3] = "Test"
        [4] = "File"

    dc:format = "image/jpeg"
    dc:creator (0x600 : isOrdered isArray)
        [1] = "Author Name"
```

prefix — Schema URI — option bits — simple property

The callback function that you provide for `DumpObject()` must have the following signature:

```
XMP_Status MyCallBack(void *callerData, XMP_StringPtr buff, XMP_StringLen buffSize){
    // Processing here
}
```

To pass your callback function to `DumpObject()`:

```
meta.DumpObject( MyCallback, &callerData );
```

The second argument is a pointer to any data type you wish, for example a file or an output stream. The data is passed as the first argument to your callback function. Your callback must cast it to the correct data type. For example, the following code demonstrates how to dump an XMP object to a file:

```
XMP_Status MyCallBack( void *callerData, XMP_StringPtr buff, XMP_StringLen buffSize ){
    XMP_Status status = 0;
    ofstream * outFile = static_cast<ofstream*>(callerData);
    (*outFile) write( buff, buffsize );
    return status;
}
// ...
std::ofstream outFile;
outFile.open( "MyFile.txt", ios::out );
meta.DumpObject( MyCallBack, &outFile );
outFile.close();
```

This callback writes the `XMP_StringPtr`, passed as *buff*, to the file, passed in as **callerData*. The callback also accepts the length of the data buffer as an `XMP_StringLen`, *buffSize*. You should make use of this to restrict that amount of data written each time the function is called.

Your callback must return a valid status; anything other than a no-error status causes output to be aborted.

Modifying XMP data in the XMP object

This section discusses how to use XMPCore to modify XMP properties. These techniques are illustrated using examples from the tutorial provided with the SDK; see "Walkthrough 2: Modifying XMP" on page 69.

NOTE ON HANDLING NEWLINES IN USER INTERFACES: The way a user interface handles newlines in text values is important to the global and cross-platform portability of XMP. When displaying text, applications should recognize common newline characters and sequences and ensure that they display as such. One technique is to modify the displayed text, substituting appropriate local newlines. You must take care, however, that the stored XMP value is not modified simply as a result of display.

Typical newlines are a single ASCII linefeed (LF, U+000A), a single ASCII carriage return (CR, U+000D), or ASCII CR-LF. Section 2.11 of the XML 1.0 specification includes other sequences as recognized newlines for normalization purposes: U+0085, U+2028, and the pair U+000D U+0085.

It is recommended that applications store all newlines in XMP text values as ASCII linefeed.

Creating and modifying simple properties

The simplest way to create a new property or set the value of an existing property is with `SXMPMeta::SetProperty()`. You provide the namespace URI, the property name, the updated value for the property; if you are creating a new property, you must also provide option flags that describe the property type, such as struct or array.

```
meta.SetProperty( kXMP_NS_XMP, "CreatorTool", "My Application Name", NULL );
```

If the property does not exist, it is created and assigned the given value. The automatic creation of properties may not always be desirable; to avoid it, use `DoesPropertyExist()` to determine if the property is already available:

```
if(meta.DoesPropertyExist( kXMP_NS_XMP, "CreatorTool" ){
    meta.SetProperty( kXMP_NS_XMP, "CreatorTool", "My Application Name", NULL )
}
```

Parameters and return values

Most of the functions in the XMP Toolkit SDK store retrieved values in buffers passed to the function (call-by-reference). The actual return value of the function, in most cases, is simply a Boolean value that describes the success of the operation.

When an operation is unsuccessful, the value of a variable passed in to store a retrieved value is not guaranteed; it may be empty, or contain the prior value, or something else. Your code must not depend on the value.

For example, this shows a common pattern for working with returned references:

```
SXMPMeta meta;

//some file operations here to get the metadata into the xmp object

std::string value;
XMP_OptionBits opts;

// get property if present:
bool result=meta.GetProperty( kXMP_NS_XMP, "CreatorTool", &value, &opts );
```

```
      // if the property is not present, the operation fails
      if (result==false)
          // "value" could contain anything, so we explicitly set it
          // with an empty string
          value.clear();

// ... continue processing...
```

Creating and modifying arrays

Use `SXMPMeta::SetArrayItem()` to modify the item values in an existing array. The function signature is similar to that of `SetProperty()`, with the addition of the index to the item that needs to be modified. Array indexes are 1-based; that is, the first item is at index 1, not 0. For example this sets the value of the first element of the 'creator' array:

```
meta.SetArrayItem( kXMP_NS_DC, "creator", 1, "Authors Name", NULL );
```

You can only use `SetArrayItem()` if the array already exists; if it does not, the function throws an exception. To create a new array or add an item to an array, use `SXMPMeta::AppendArrayItem()`.

To create a new array, you must supply options flags that describe the type of the array; ordered, unordered, or language-alternative. If you do not supply the options flags, and the array does not already exist, `AppendArrayItem()` throws an exception. For example:

```
// Will throw an exception if array does not exist
meta.AppendArrayItem( kXMP_NS_DC,"creator",0,"A Name",NULL );
```

If the XMP already has a property from the Dublin Core schema named 'creator', the array is modified; an item is added to the array with value of 'A Name'. However, because no options flag is supplied, if the array does not exist, the function throws an exception and does not create a new array.

```
// Create the array if it does not already exist
meta.AppendArrayItem( kXMP_NS_DC,"creator",kXMP_PropArrayIsOrdered,"A Name",NULL );
```

Here again, if the XMP already has the property, the existing array is modified. However, because the options flag is supplied, if the property does not exist, it is created as an ordered array.

Modifying and creating complex properties

Schemas can contain very complex properties, such as arrays nested within structures. A complex property can be, for instance:

▶ A structure with nested arrays.

▶ A structure with nested structures.

▶ An array with nested structures.

▶ An array with nested arrays.

Each property can have an arbitrary number of nested levels. For example, a structure can have arrays as its field values; the array items themselves could also be arrays or structures, and so on.

This figure shows a conceptual diagram of a complex property:
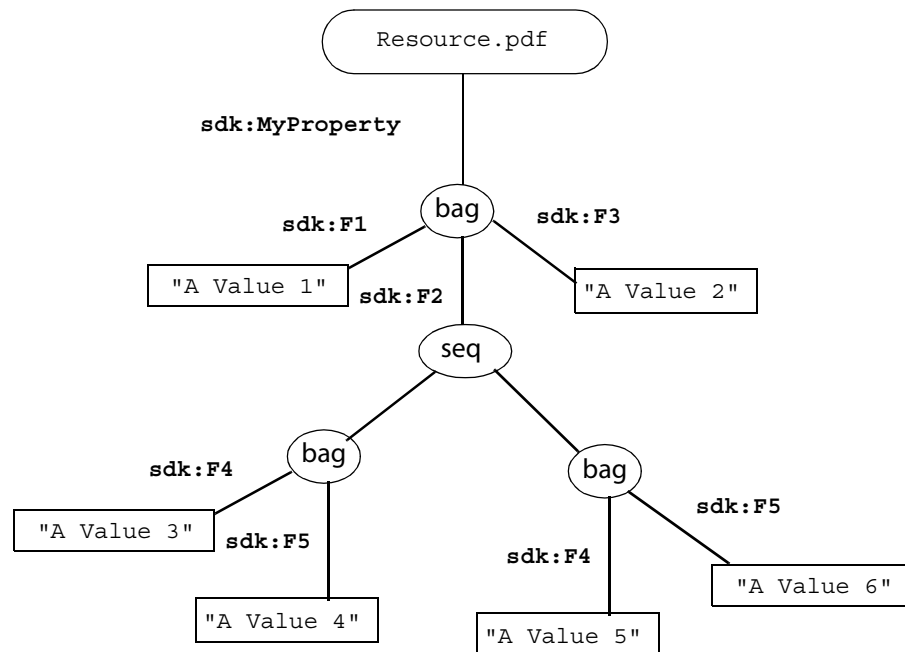
In this example, `Resource.pdf` has a single property named `"MyProperty"`. The property type is a structure which has several fields, one of which is an ordered array. The array holds items which are themselves structures.

The debugging dump of this XMP object looks like this:

```
sdk: http://ns.adobe.com/xmp/sdk/ (0x80000000 : schema)
    sdk:MyProperty (0x100 : isStruct)
        sdk:F1 = "A Value1"
        sdk:F2 (0x600 : isOrdered isArray)
            [1] (0x100 : isStruct)
                sdk:F4 = "A Value3"
                sdk:F5 = "A Value4"
            [2] (0x100 : isStruct)
                sdk:F4 = "A Value5"
                sdk:F5 = "A Value6"
        sdk:F3 = "A Value2"
```

The same XMP serialized to RDF looks like this:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about=""
      xmlns:sdk="http://ns.adobe.com/xmp/sdk/">

    <sdk:MyProperty rdf:parseType="Resource">
        <sdk:F1>A Value1</sdk:F1>
```

```
        <sdk:F2>
            <rdf:Seq>
                <rdf:li rdf:parseType="Resource">
                    <sdk:F4>A Value3</sdk:F4>
                    <sdk:F5>A Value4</sdk:F5>
                </rdf:li>
                <rdf:li rdf:parseType="Resource">
                    <sdk:F4>A Value5</sdk:F4>
                    <sdk:F5>A Value6</sdk:F5>
                </rdf:li>
            </rdf:Seq>
        </sdk:F2>

        <sdk:F3>A Value2</sdk:F3>
    </sdk:MyProperty>

  </rdf:Description>
</rdf:RDF>
```

In order to create properties like this, you must provide the correct name for the property when using any of the property-setting functions of `SXMPMeta`. To access deeply nested properties, such as `sdk:F5`, the name alone does not point to the correct property; you must provide a path. For example:

```
MyProperty2/sdk:F2[last()]/sdk:F5
```

—or—

```
MyProperty2/sdk:F2[2]/sdk:F5
```

You should not try to compose a complex path by hand. The XMP API provides utility functions, which you should use to compose paths to deeply nested properties.

Composing paths to complex properties

The functions in `SXMPMeta` such as `GetProperty()`, `GetArrayItem()`, and `GetStructField()` provide easy access to top-level simple properties, items in top-level arrays, and fields of top-level structs. They are not as convenient for more complex properties, such as fields several levels deep in a complex struct. The `SXMPUtils` class provides path composition functions that help you access such complex properties.

All of the utility functions are static; that is, they are called directly from the `SXMPUtils` class. You never need to instantiate the `SXMPUtils` class.

You can create the complex properties by composing the paths using the utility function, then setting property values incrementally.

For example, to add a value to `sdk:F4` in the complex property structure shown in the diagram above, you would first create the `MyProperty` structure, using the options flag `kXMP_PropValueIsArray` to signify the value type of the property.

The structure `sdk:F2` contains three fields, the second of which is an ordered array. To create the array, you would follow these steps, illustrated by the following code extract:

1. Compose a path to the second field of the structure, by using the utility function `SXMPUtils::ComposeStructFieldPath()`.

2. Create the array by passing the composed path to `SXMPMeta::AppendArrayItem()`.

3. Compose a path to the element that has just been added, using
    `SXMPUtils::ComposeArrayItemPath()`.

4. Compose a path to the structure field named `sdk:F4`.

5. Set the value for the structure field `sdk:F4`.

```
string ns = "http://ns.adobe.com/xmp/sdk/";
meta.SetProperty( ns.c_str(), "MyProperty", NULL, kXMP_PropValueIsStruct );
string path;

SXMPUtils::ComposeStructFieldPath( ns.c_str(), "MyProperty",
    ns.c_str(), "F2", &path);
// path is now "MyProperty2/sdk:F2"

meta.AppendArrayItem( ns.c_str(), path.c_str(),
    kXMP_PropArrayIsOrdered, NULL, kXMP_PropValueIsStruct );

SXMPUtils::ComposeArrayItemPath( ns.c_str(), path.c_str(),
    kXMP_ArrayLastItem, &path );
// path is now "MyProperty2/sdk:F2[last()]"

SXMPUtils::ComposeStructFieldPath( ns.c_str(), path.c_str(),
    ns.c_str(), "F4", &path );
// path is now "MyProperty2/sdk:F2[last()]/sdk:F4"

meta.SetProperty( ns.c_str(), path.c_str(), "AValue3", NULL );
```

This illustrates the general technique; you can, of course, use different functions to achieve the same effect. In this case, for instance, you could use the function `SXMPMeta::SetStructField()`, replacing the last two lines of code with the following:

```
meta.SetStructField( ns.c_str(), path.c_str(),
    ns.c_str(), "F4", "aValue3", NULL );
```

It is sometimes possible, depending on the property type, to create several nested properties at once, just by providing a path expression. For example, to create a structure of structures, you could supply the path and have the properties created directly. The following code adds several nested structures to a new property:

```
SXMPUtils::ComposeStructFieldPath( ns.c_str(), "MyProperty", ns.c_str(),
    "StructOne", &path );
SXMPUtils::ComposeStructFieldPath( ns.c_str(), path.c_str(), ns.c_str(), "
    StructTwo", &path );
SXMPUtils::ComposeStructFieldPath( ns.c_str(), path.c_str(), ns.c_str(),
    "StructThree", &path );

meta.SetProperty( ns.c_str(), path.c_str(), "MyValue", NULL );
```

The structures are not created explicitly by the path composition functions, but are added when the path is evaluated in the `SetProperty()` call.

Modifying qualifiers in complex properties

Leaf properties of complex structs and arrays can have property qualifiers and text alternatives, regardless of how deeply nested they may be. The arrays and structures themselves cannot have qualifiers; see the *XMP Specification Part 1, Data and Serialization Models*.

To add a qualifier to a property, you must provide the name of the qualifier and the name of the property to which it is attached. In the case of a property within a nested structure, you must compose the path to

the field and then set the qualifier value. The following shows how to add a qualifier for `sdk:F4` from the example:

```
string path;
SXMPUtils::ComposeStructFieldPath( ns.c_str(), "MyProperty",
    ns.c_str(), "F2", &path );

SXMPUtils::ComposeArrayItemPath( ns.c_str(), path.c_str(), 1, &path );
SXMPUtils::ComposeStructFieldPath( ns.c_str(), path.c_str(),
    ns.c_str(), "F4", &path );

meta.SetQualifier( ns.c_str(), path.c_str(), ns.c_str(), "MyQual", "MyValue" );
```

To set the qualifier with a binary value, compose the path using `SXMPUtils::ComposeQualifierPath()` and use the appropriate mutator for setting the binary value:

```
SXMPUtils::ComposeStructFieldPath( ns.c_str(), "MyProperty",
    ns.c_str(), "F2", &path );

SXMPUtils::ComposeArrayItemPath( ns.c_str(), path.c_str(), 1, &path );

SXMPUtils::ComposeStructFieldPath( ns.c_str(), path.c_str(),
    ns.c_str(), "F4", &path );

SXMPUtils::ComposeQualifierPath( ns.c_str(), path.c_str(),
    ns.c_str(), "MyBoolQual", &path );

meta.SetProperty_Bool( ns.c_str(), path.c_str(), true, NULL );
```

Modifying language alternatives

Use `SXMPMeta::SetLocalizedText()` to modify an alt-text array. The function signature is:

```
bool TXMPMeta< tStringObj >::SetLocalizedText (
    XMP_StringPtr schemaNS,
    XMP_StringPtr altTextName
    XMP_StringPtr genericLang,
    XMP_StringPtr specificLang,
    tStringObj * itemValue,
    XMP_OptionBits * options )
```

The *specificLang* and *genericLang* arguments determine which item in the array is modified. The result depends on whether the array is empty and if there is an x-default item present:

▶ If the array is empty (that is, there are no items in the array, including an x-default item) then two items are added to the array.

  ▷ An x-default item is created at index 1 with the value of *itemValue*, and with an `xml:lang` property qualifier with the value of 'x-default'.

  ▷ An item is added to the array at the first available index (in this case 2) with a value of *itemValue* and with an `xml:lang` property qualifier with the value of *specificLang*.

▶ If the array only contains an x-default item, the x-default item is modified and a new item added to the array.

  ▷ The x-default item value is set to *itemValue* and has an `xml:lang` qualifier with a value of 'x-default'.

▷   An item is added to the array at the first available index (in this case 2) with a value of *itemValue* and with an `xml:lang` property qualifier with the value of *specificLang*.

▶   If there are already several items in the array, including the x-default item:

▷   If an item matches *specificLang*, that item's value is modified. If the matching item's existing value matches that of the x-default item, the x-default item is also modified.

▷   If an item matches the *genericLang* and there are no other items that match the *genericLang*, that item's value is modified. If the item's existing value matches that of the x-default item then the x-default item is also modified.

▷   If an item matches the *genericLang* and there are other items that match the *genericLang*, a new item is created. The new item value is set to *itemValue* and given an `xml:lang` qualifier with the value of *specificLang*.

If you want to always add an item to the array if it does not already exist, then do not use both the *genericLang* and the *specificLang* arguments. If you specify only a specific language, you guarantee that a new item is added if an exact match is not made.

Deleting language alternatives

Use `SXMPMeta::DeleteLocalizedText()` to delete specific language alternatives from an alt-text array. The function signature is:

```
bool TXMPMeta< tStringObj >::DeleteLocalizedText (
    XMP_StringPtr schemaNS,
    XMP_StringPtr altTextName
    XMP_StringPtr genericLang,
    XMP_StringPtr specificLang)
```

The rules for finding the language value to delete, based on the `genericLang` and `specificLang` values, are similar to those for `SetLocalizedText()`.

Accessing language alternatives in complex properties

To create an alt-text array within a complex property, construct the correct path to the property, using the `SXMPUtils` path composition functions. You can then use `SetLocalizedText()` to add items.

This code adds an alt-text array and several items to the complex structure in the example shown in "Modifying and creating complex properties" on page 20. The new property has the name `F6`:

```
SXMPUtils::ComposeStructFieldPath(ns.c_str(), "MyProperty",
    ns.c_str(), "F2", &path);

SXMPUtils::ComposeArrayItemPath( ns.c_str(), path.c_str(), 1, &path );

SXMPUtils::ComposeStructFieldPath( ns.c_str(), path.c_str(),
    ns.c_str(), "F6", &path );

meta.SetLocalizedText( ns.c_str(), path.c_str(), "en", "en-US", "Color", NULL );
meta.SetLocalizedText( ns.c_str(), path.c_str(), "", "en-GB", "Colour", NULL );
```

There is also a utility function for composing a path to a desired item within an alt-text array based on the value of the property qualifier. `SXMPUtils::ComposeLangSelector()` composes a path to a specific item, and only that item.

For example, the following shows XMP containing some alt-text items:

```
sdk: http://ns.adobe.com/xmp/sdk/ (0x80000000 : schema)
   sdk:MyProperty (0x100 : isStruct)
       sdk:F1 = "A Value1"

       sdk:F2 (0x600 : isOrdered isArray)
           [1] (0x100 : isStruct)
               sdk:F4 = "A Value3" (0x10 : hasQual)
                   ? sdk:MyQual = "MyValue" (0x20 : isQual)
                   ? sdk:MyBoolQual = "True" (0x20 : isQual)
               sdk:F5 = "A Value4"

               sdk:F6 (0x1E00 : isLangAlt isAlt isOrdered isArray)
                   [1] = "Color" (0x50 : hasLang hasQual)
                       ? xml:lang = "x-default" (0x20 : isQual)
                   [2] = "Color" (0x50 : hasLang hasQual)
                       ? xml:lang = "en-US" (0x20 : isQual)
                   [3] = "Colour" (0x50 : hasLang hasQual)
                       ? xml:lang = "en-GB" (0x20 : isQual)
           [2] (0x100 : isStruct)
               sdk:F4 = "A Value5"
               sdk:F5 = "A Value6"
       sdk:F3 = "A Value2"
```

To access the GB English item and only that item, use this code:

```
string gbItemPath, gbItemVal;
SXMPUtils::ComposeLangSelector( ns.c_str(), path.c_str(), "en-GB", &gbItemPath );
// gbItemPath = MyProperty/sdk:F2[1]/sdk:F6[?xml:lang="en-GB"]
bool exists = meta.GetProperty( ns.c_str(), gbItemPath.c_str(), &gbItemVal, NULL );
```

The difference between these two function calls is subtle:

▶   `ComposeLangSelector()` returns a path that accesses a specific item, and only that item.

▶   The `Get/SetLocalizedText()` functions access the best matching item for the language parameters, and if all else fails, return or set the `x-default` item.

Modifying dates and times

Use `SXMPMeta::SetProperty_Date()` to modify properties with date-time values. You can use the utility functions in `SXMPUtils` to obtain or construct an `XMP_DateTime` object to represent any date and time. For example, to apply the current date and time to a property:

```
XMP_DateTime updatedTime;
SXMPUtils::CurrentDateTime( &updatedTime );
meta.SetProperty_Date( kXMP_NS_XMP, "MetadataDate", updatedTime, NULL );
```

`SXMPUtils` provides a number of utility functions for examining, creating, and manipulating dates; for example, to convert between time-zone and UTC times. The utility function `SXMPUtils::ConvertFrom_Date()` converts an `XMP_DateTime` value into a string formatted according to the ISO 8601 profile at http://www.w3.org/TR/NOTE-datetime. For example, *2006-04-10T16:35:34+0100*.

You can use `SXMPUtils::CompareDateTime()` to compare date-time instances; for example, if you have several dates and want to apply the latest time to a property. This shows how to compare dates, and also how to construct a date-time value from an array:

```
XMP_DateTime myDate = { 2000, 4, 8, 18, 22, 57, 0, 0, 0, 0 };
XMP_DateTime timeNow;
SXMPUtils::CurrentDateTime( &timeNow );

int result = SXMPUtils::CompareDataTime( myDate, timeNow );

if( result < 0 )
    // myDate is before timeNow
else if( result > NULL )
    // timeNow is before myDate
else
    // dates are equal
```

Using local time values

It is recommended that you use local times, with a time zone designator of `+hh:mm` or `-hh:mm`, instead of `Z`. This promotes human readability. For example, for a file saved in Los Angeles at 10 pm on July 23, 2005, a timestamp of `2005-07-23T22:00:00-07:00` is understandable, while `2005-07-24T05:00:00Z` is confusing.

Working with schemas

There are a range of metadata schemas available for you to take advantage of, as described in the *XMP Specification Part 2, Standard Schemas*. If you need to, however, you can either extend an existing schema or create a new one.

Although you can add new properties to extend an existing schema, the standard schemas are generally intended for specific uses and should not be altered. Although you can, technically, add a property to, say, the Dublin Core schema, it is not recommended. If you need a specific set of properties, you should create a new schema. For example, you might create a schema `com.`*`companyName`*`.xmp/1.0`, and properties within that schema for `itemCode`, `orderNumber`, and so on.

Creating custom schemas

To create a schema, you define your own namespace and within that namespace you define a set of properties. Once you have created a schema, you can extend it by adding new properties at any time; see .

A schema must have a unique name, in order to avoid collisions with properties in other schemas. The schema name is in the form of a URI and must comply with the XML 1.0 namespace rules, as defined at http://www.w3.org/TR/2006/REC-xml-names-20060816/. You can also define a preferred prefix to use with your namespace; defining a prefix is not mandatory, but it is recommended.

You will add properties to your namespace as you would to an existing schema. For your own schema, you should also create a *specification document* that lists and describes all of the properties and data types. The specification document is plain text, human readable, and does not need to be in any specialized format. You should make it available to anyone wishing to work with your custom schema; however, it is not necessary to publish it in the public domain.

Registering namespaces

To use your custom schema, you must register the namespace and prefix that you have chosen. There are two ways to do this:

▶ Register the namespace explicitly using the static function `SXMPMeta::RegisterNameSpace()`.

▶ Create a new XMP object from valid RDF. Unknown namespaces are registered automatically.

To register a namespace explicitly, you provide the namespace URI and a preferred prefix:

```
string actualPrefix
SXMPMeta::RegisterNamespace( "http://ns.adobe.com/MyNamespace/",
    "MyPrefix", &actualPrefix );
```

The prefix you pass in is a suggestion; it is not guaranteed to be registered as the prefix for the namespace. The last argument, *actualPrefix*, returns the actual prefix that will be used for the registered namespace.

▶ If you register a new namespace with a new prefix (that is, one not yet in use), the namespace is registered with your suggested prefix, and the function returns it in the *actualPrefix* buffer.

▶ If the prefix you supply for a new namespace is already in use, the function generates a new, unique prefix based on the one you supplied. For example, if there is already a prefix "MyPrefix" then the actual prefix returned is "MyPrefix_1_"; or, if "MyPrefix_1_" is already used, "MyPrefix_2_", and so on.

▶ If you register an existing namespace with a new prefix, the function returns the prefix that is already registered for that namespace.

The same rules apply if you create an XMP object from an RDF stream.

Prefix collisions can occasionally occur, both at run time and when XMP is serialized and stored. You should never depend on the suggested prefix being used, but always check for and use the actual returned prefix when registering a namespace.

To view all of the registered namespaces and their prefixes you can create a dump using a callback function. See for details on using callbacks and debugging dumps.

Extending schemas

There are no restrictions as to what properties you can add to a namespace; however, it is recommended that you try to keep properties consistent with those already existing in a schema.

▶ New properties that you add to a schema do not interfere with existing applications, as they have no knowledge of your extensions. This also means, of course, that they cannot take advantage of them. Your extensions are of use only to your own applications.

▶ You should not change existing property definitions from other schemas. This may cause existing applications to perform unexpectedly and produce incorrect results.

To add a new property to a schema you need only create that property and set a value, even if the value is empty. For example if you wish to add a new text property to a schema you have created, the following is sufficient:

```
meta.SetProperty( myNamepaceURI, "MyNewProperty", "The property value", NULL );
```

Any property type can be used when extending a schema and there are is no limitation to the number of properties that can be added. When the XMP object is written back to the resource, the new property is also written.

As long as you have provided a unique namespace URI for your schema, new properties that you define are guaranteed to not interfere with existing schemas or their properties, even if your new properties have the same local name as a property defined elsewhere.

You can also define new property value types; for example, your properties can define structures and arrays. The specification document for your schema should document all schema properties and any new property types.

Iterating over metadata

The XMP Toolkit SDK API provides the `SXMPIterator` class which allows you to traverse over an XMP object. The iterator allows you to define a starting point for the iteration and what properties should be visited.

Creating iterators

There are several constructors available for the iterator object that provide different levels of control over how the iteration is done. The simplest iterator allows you to traverse an entire XMP object:

```
SXMPIterator iter( meta );
```

If you do not need to traverse the entire tree, you can provide the constructor with a schema name to control where the iterator starts. To traverse a single schema, simply specify the schema name. For example, this iterator only visits nodes from the Dublin Core schema:

```
SXMPIterator iter( meta, kXMP_NS_DC );
```

You can further specify a property within the schema at which to start the iteration. This is particularly convenient for complex properties that may have many nested levels of arrays and structures. This code creates an iterator for the single property `"`"Keywords`"` in the XMP Basic schema:

```
SXMPIterator iter( meta, kXMP_NS_XMP, "Keywords" );
```

Finally, you can provide option bits that control how the iteration progresses from the starting point. If, for example, you do not want to traverse the entire subtree in a schema, you can limit the iteration to a single level with the options `kXMP_IterJustChildren`:

```
SXMPIterator iter( meta, kXMP_NS_DC, kXMP_IterJustChildren );
```

This iterator visits only the immediate children of the root node, which in this case is the Dublin Core schema. You can provide the options flag as the last parameter, following any other set of parameters (that is, XMP object only; XMP object and schema; or XMP object, schema, and property).

A number of options flags (`kXMP_Iter*`) allow different kinds of control over which nodes are visited during the iteration. You can, for example, create an iterator that visits only leaf nodes:

```
SXMPIterator iter( meta, kXMP_NS_DC, kXMP_IterJustLeafNodes );
```

This iterator visits only nodes that can have values. For example if you have a structured property with several fields, the iterator does not visit the structure node itself, but does visit all of the structures fields.

You can combine multiple option bits with a logical OR to create the desired iterator. See for the list of available options.

Visiting nodes

To use your iterator instance, call its `Next()` method. The first time you call `Next()`, the method provides information for the first node; each succeeding call visits the next node, until the iteration completes. The method returns a Boolean value to indicate whether there are more nodes to visit; true if there are, and false if the iteration is complete.

The parameters you pass to the `Next()` method store the details of the current node; that is, the schema name, the property path, the property value, and the option bits that describe the property. All of the parameters are optional; if you don't need all of the information, you can omit or pass `null` for any of them. For example, the following displays the schema name, path, and value of all properties in the Dublin Core schema of the XMP object, but does not collect the property-type options flag:

```
SXMPIterator iter( meta, kXMP_NS_DC );

while( iter.Next( &schemaNS, &propPath, &propVal )){
    cout << schemaNS << "  " << propPath << " = " << propVal << endl;
}
```

The default behavior of the iterator is to return a full path to the property; however, with complex properties you may not be interested in the full path, but only in the property name. In this case, you can create an iterator that uses the options flag `kXMP_IterJustLeafName`. This affects how the path is returned for a visited node.

For example, the following shows an RDF extract representing a complex structure:

```
<xmpTest:MyTopStruct rdf:parseType="Resource">
    <xmpTest:MySecondStruct rdf:parseType="Resource">
        <xmpTest:MyThirdStruct rdf:parseType="Resource">
            <xmpTest:MyThirdStructField>Field Value 3</xmpTest:MyThirdStructField>
        </xmpTest:MyThirdStruct>
    <xmpTest:MySecondStructField>Field Value 2</xmpTest:MySecondStructField>
    </xmpTest:MySecondStruct>
    <xmpTest:MyTopStructField>Field Value 1</xmpTest:MyTopStructField>
</xmpTest:MyTopStruct>
```

This code creates an iterator that returns only the name of the property and not the full path for this structure:

```
XMP_OptionBits opts = ( kXMP_IterJustLeafNodes | kXMP_IterJustLeafName );
SXMPIterator iter( meta, kXMP_NS_SDK, "MyTopStruct", opts );

while( iter.Next( &schemaNS, &propPath, &propVal )){
    cout << propPath << " = " << propVal << endl;
}
```

The following shows how the returned string can differ significantly depending on the options used to create the iterator. This shows output from two different iterators for the same XMP object that visit the same nodes, but use different output options:

In this example, only the fields with values have been visited, because the option flag
`kXMP_IterJustLeafNodes` was used to construct both iterators. The first set of output lines shows the
complete path for each field, which is the default behavior. The second set of output lines shows only the
property name because the option flag `kXMP_IterJustLeafName` was used.

Skipping nodes

The iterator object has a `Skip()` method that allows you to skip some or all of the remaining iterations.
The function accepts option bits that describe what should be skipped. In the following call, the option
`kXMP_IterSkipSubtree` skips a subtree below the current node:

```
iter.Skip( kXMP_IterSkipSubtree );
```

For example, if you want your iterator to visit each node in the XMP data tree except a particular schema
and then continue visiting other nodes in other schemas, use something like this:

```
SXMPIterator skipExifIter ( meta ); // Visit all nodes and properties

while( skipExifIter.Next( &schemaNS, &propPath, &propVal )){
    if( schemaNS == kXMP_NS_EXIF ){
        skipExifIter.Skip( kXMP_IterSkipSubtree );

    } else {
        cout << schemaNS << "  " << propPath << " = " << propVal << endl;
    }
}
```

The options `kXMP_IterSkipSiblings` is similar, but with a subtle difference. Like
`kXMP_IterSkipSubtree`, it skips any subtree below the current node; however, it also skips any siblings of
the current node. To understand the effect, consider what the current node is and where it appears in the
XMP data tree:

▶   If the current node is a top-level node and you skip all of its siblings, the iteration is complete and no
    more nodes will be visited.

▶   If the current node is nested some number of levels down, then other nodes higher up the tree will still
    be visited.

To illustrate this, consider the metadata structure in the following illustration. When the iteration reaches
"Item 4", we want to skip the rest of the array, but go on to visit remaining nodes at a higher level.

To achieve this behavior, use code like this:

```
XMP_OptionBits opts = kXMP_IterJustLeafNodes | kXMP_IterOmitQualifiers |
    kXMP_IterJustLeafName;
SXMPIterator skipIter ( meta, kXMP_NS_SDK, opts );

while( skipIter.Next( &schemaNS, &propPath, &propVal )){
    if( propVal == "Item 4" ){
        skipIter.Skip( kXMP_IterSkipSiblings );

    } else {
        cout << schemaNS << "  " << propPath << " = " << propVal << endl;
    }
}
```

If "Item 4" were a complex property, rather than a simple one, this would skip any subtree below that node, *and* any of that node's remaining siblings (the rest of the items in "NestedArray").
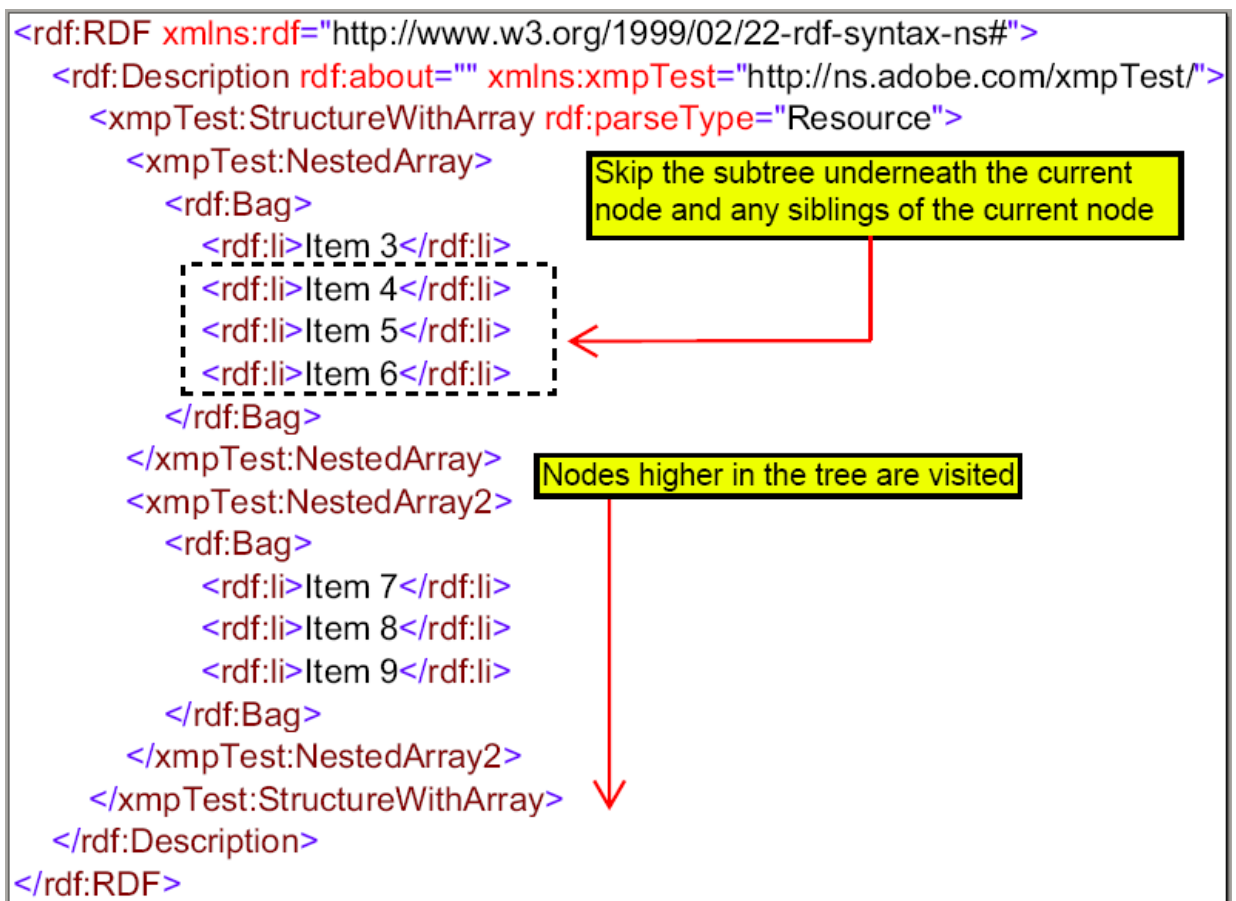
API summary: the XMPCore component

The client view of the XMPCore API is provided through three C++ class templates:

▶ `TXMPMeta` provides the core services of the XMP Toolkit SDK. Its methods allow you to create and delete metadata properties, and to retrieve and modify property values.

▶ `TXMPUtils` provides additional utilities layered on top of `TXMPMeta`.

▶ `TXMPIterator` provides methods to iterate over existing XMP metadata.

In your code, you will work with the concrete classes `SXMPMeta`, `SXMPUtils`, and `SXMPIterator`.

Instantiate the `SXMPMeta` class to represent a set of metadata. You can create an empty object and fill it with a string of serialized XMP data that you construct, or you can read the XMP metadata from a file into the object. Use the `SXMPMeta` functions to work with namespaces and properties.

▶ You can create your own private namespaces, but must register them before use, using `SXMPMeta::RegisterNamespace()`.

▶ Property accessor functions of various kinds allow you to retrieve and set existing property values, and to create new properties.

   ▷ You can get and set property values as strings, or as binary types. `SXMPUtils` provides functions for converting between types.

   ▷ Use `SXMPMeta::Get/SetLocalizedText()` with language alternative (alt-text) arrays.

   ▷ Use the path composition functions provided by `SXMPUtils` to construct complex paths with nested structs or arrays. It is recommended that you not depend on specific namespace prefixes; rather than hard-coding the prefixes for struct fields, use functions like `GetStructField()` and `SetStructField()`, or `ComposeStructFieldPath()`.

▶ Create an `SXMPIterator` object to operate on all or a subset of the properties in the metadata tree contained in an `SXMPMeta` object.

There is an additional constant class `XMP_DateTime`, which is not provided as a template, but is simply defined. See the API documentation for details.

SXMPMeta class

This class defines functions for working with namespaces and with properties, and for serializing XMP data into RDF text. In addition, static functions available through the `SXMPMeta` class allow you to initialize and terminate the Toolkit cleanly, discover the Toolkit version, set up user notification for Toolkit failures, and write out debugging information.

▶ "Creating metadata objects" on page 33

▶ "Preparing metadata for I/O" on page 34

▶ "Working with namespaces" on page 36

▶ "Working with properties" on page 37

▶ "Toolkit configuration" on page 40

Creating metadata objects

The default constructor creates a new empty `SXMPMeta` object:

```
SXMPMeta ();
```

Another form of the constructor allows you to create a new `SXMPMeta` object and populate it with metadata from a buffer containing serialized RDF text.

```
SXMPMeta (ptrToXML, sizeOfXML);
```

A buffer of RDF text that you pass directly to the constructor must contain a complete XMP packet. See for additional options.

Copying metadata

You can pass an existing `SXMPMeta` object or reference to the constructor to do a shallow copy of the object:

▶ Create a new `SXMPMeta` object that refers to the same internal XMP object as an existing `SXMPMeta` object.

▶ Create a new `SXMPMeta` object that refers to the underlying reference for an existing XMP object, which was obtained from some other XMP object by the `GetInternalRef()` method. This is used to safely pass XMP objects across DLL boundaries.

These operations increment the reference count for the object, but do not perform a deep copy. They return a reference to the same underlying object; they do not create a new object.

To reproduce an entire metadata tree contained in an XMP object, use `SXMPMeta::Clone()`. The clone function returns a new object, not a pointer to the same object. The `Clone()` function is easy to misuse. The following examples show correct and incorrect usage:

CORRECT USAGE: In this example, the `clone2` line shows that you do not have to use an explicit pointer. This is good for local usage, you do not have to worry about memory leaks.

```
SXMPMeta * clone1 = new SXMPMeta( sourceXMP.Clone() ); // This works
SXMPMeta clone2( sourceXMP.Clone() ); // This works also. (Not a pointer)
```

INCORRECT USAGE: In this code, the assignment to `clone3` creates a temporary object, initializes it with the clone, assigns the address of the temporary to `clone3`, then deletes the temporary.

```
SXMPMeta * clone3 = &sourceXMP.Clone(); // This does not work!
```

Preparing metadata for I/O

To store and retrieve XMP metadata, it must first be serialized into XML text (specifically a subset of RDF). The serialization protocol is described in *XMP Specification Part 1, Data and Serialization Models*. The serialized data is then wrapped in packets for embedding in files; the *XMP Specification Part 3, Storage in Files*, describes the structure and capabilities of these packets.

▶ The function `SXMPMeta::SerializeToBuffer()` transforms the metadata tree contained in the XMP object into a buffer of RDF text that conforms to the serialization model (see).

▶ The serialized buffer that results from the call to `SXMPMeta::SerializeToBuffer()`, or a serialized buffer that you read from a file using `XMPFiles::GetXMP()` (see) can be reconstituted as an `SXMPMeta` object using `SXMP::ParseFromBuffer()`. You can also parse multiple buffers of partial data into a single XMP object (see).

These functions support serialization of metadata into RDF text that conforms to the serialization model given in the *XMP Specification Part 1, Data and Serialization Models*.

`SerializeToBuffer()`	Serializes an XMP object into a buffer as RDF that conforms to the serialization model.
	You can provide option flags that control how serialization is performed.
`ParseFromBuffer()`	Parses RDF from one or more input buffers into an `SXMPMeta` object. The input for parsing can be any valid Unicode encoding.
	The buffers can be any length. The buffer boundaries need not respect XML tokens or even Unicode characters.

Serializing for output

To specify options for serialization, use a logical OR of these bit-flag constants:

> **`kXMP_OmitPacketWrapper`**: Do not include an XML packet wrapper.
> **`kXMP_ReadOnlyPacket`**: Create a read-only XML packet wapper.
> **`kXMP_UseCompactFormat`**: Use a highly compact RDF syntax and layout.
> **`kXMP_ExactPacketLength`**: The padding parameter provides the overall packet length. The actual amount of padding is computed. An exception is thrown if the packet exceeds this length with no padding.

The options flags that you specify for serialization must be logically consistent; if they are not, an exception is thrown. You cannot specify `kXMP_OmitPacketWrapper` along with `kXMP_ReadOnlyPacket`, or `kXMP_ExactPacketLength`.

In addition, you can include one encoding option flag:

> **`kXMP_EncodeUTF8`**: Encode as UTF-8, the default.
> **`kXMP_EncodeUTF16Big`**: Encode as MSB-first (big-endian) UTF-16.
> **`kXMP_EncodeUTF16Little`**: Encode as LSB-first (little-endian) UTF-16.
> **`kXMP_EncodeUTF32Big`**: Encode as MSB-first (big-endian) UTF-32.
> **`kXMP_EncodeUTF32Little`**: Encode as LSB-first (little-endian) UTF-32.

For an example, see .

Parsing serialized data into an XMP object

You can obtain a buffer of serialized metadata from an existing `SXMPMeta` object using `SXMPMeta::SerializeToBuffer()`, or from a file, using `SXMPFiles::GetXMP()`, or you might obtain or construct the buffer in some other way, as long as it conforms to the serialization model given in the *XMP Specification Part 1, Data and Serialization Models*.

A buffer that contains a complete XMP packet can be reconstituted as an `SXMPMeta` object simply by passing it to the constructor:

```
SXMPMeta( ptrToXML, sizeOfXML );
```

The length is the number of bytes, regardless of the character encoding.

This is the equivalent of creating an empty object, then calling `SXMPMeta::ParseFromBuffer()`:

```
SXMPMeta meta;
meta::ParseFromBuffer( ptrToXML, sizeOfXML );
```

This parses a full XMP packet. You will more typically need to parse multiple buffers containing partial data into a single `SXMPMeta` object. To do this, you can make multiple calls to `SXMPMeta::ParseFromBuffer()`, passing the RDF in a sequence of buffers, and setting the option flag `kXMP_ParseMoreBuffers` for all but the last call.

For examples, see "Parsing XMP" on page 53.

Working with namespaces

XMP uses XML namespaces for top-level properties, struct fields, and qualifiers. This is a requirement inherited from RDF. The current specification for XML namespaces is `"Namespaces in XML 1.0:"`

http://www.w3.org/TR/2006/REC-xml-names-20060816/

For important details about constructing and referencing namespace names, refer to the discussion of namespaces, prefixes, and XMP schemas in the *XMP Specification Part 1, Data and Serialization Models*.

XMPCore maintains a table of registered namespaces, which is initialized with a number of standard namespaces. You can register custom namespaces, using the function `SXMPMeta::RegisterNamespace()`. It is not an error if a specified namespace URI is already registered; the function call does nothing in this case. If XMPCore encounters an unknown namespace when parsing XML into an `SXMPMeta` object, it automatically registers the namespace.

The table of registered namespaces has one entry for each unique namespace URI, and one prefix for each URI. When you explicitly register a new namespace, you can specify a suggested prefix. If the URI is not already in the table, the suggested prefix is used only if that prefix is not already in use for some other URI. If the suggested prefix is already in use, a derived prefix is constructed by appending a numeric suffix.

It is very important to understand that prefixes are local and scoped in the XML. The prefix is only a means to look up the URI; the prefix itself is not considered in name comparison. Software must never depend on the use of a specific prefix in stored XML. See "Creating custom schemas" on page 27.

When XMPCore parses XML, the stored prefix is used to look up the URI, then the URI is looked up in the registered namespace table and added if not already there. This `"active"` prefix is unique within XMPCore. XMPCore uses only the active prefix from the registered namespace table when it serializes data to XML; a different prefix from parsed input is irrelevant.

These `SXMPMeta` functions manipulate namespaces:

`RegisterNamespace()`	Registers a namespace URI and prefix.
`GetNamespacePrefix()`	Obtains the prefix for a registered namespace URI.
`GetNamespaceURI()`	Obtains the URI for a registered namespace prefix.

Working with properties

The first two parameters of all property access functions are the top-level namespace URI (the "schema" namespace) and the basic name of the property being referenced. You can use the utility functions provided by the XMPUtils class to compose path expressions to deeply nested properties.

Option constants describe the property type, which can be a simple type, a structure, or an array, as described in the *XMP Specification Part 1, Data and Serialization Models*.

▶ Simple properties, simple array items, and simple struct fields have values. Arrays and structs do not have values. Leaf items or fields can contain values, or can be empty arrays or structs.

▶ There can be complex nesting of arrays, structs, and qualifiers. The SXMPUtils class offers functions that help construct complex paths, which you can then pass to the property accessor functions.

▶ For properties that can have values, the values can be set and retrieved either as strings or as binary values. The SXMPUtils class offers functions to help convert between these types.

The property accessors always use Unicode strings with UTF-8 encoding. When you serialize the data, you can specify other encodings; see "Serializing for output" on page 35.

Setting property values and creating properties

You can create new properties using the property-setting functions. To create empty arrays and structs, use the appropriate option flags. When you set a leaf value for a struct, all levels that are assigned implicitly are created if necessary. Similarly, adding an array item with AppendArayItem() creates the named array if necessary.

These functions set property values using UTF-8 encoded strings:

SetProperty()	The simplest property setter, mainly for top level simple properties or after using the path composition functions in XMPUtils.
SetArrayItem()	Provides access to items within an array. The index is passed as an integer, you need not worry about the path string syntax for array items, converting a loop index to a string, and so on. The specified array must already exist; to create a new array, use AppendArrayItem().
	In normal usage the selected array item is modified. A new item is automatically appended if the index is the array size plus 1. To insert a new item before or after another item, use one of the option flags:
	`kXMP_InsertBeforeItem` `kXMP_InsertAfterItem`
AppendArrayItem()	Simplifies construction of an array by not requiring that you pre-create an empty array. The array that is assigned is created automatically if it does not yet exist and the correct options are supplied; otherwise, the method throws an exception.
	Each call appends an item to the array. The corresponding parameters have the same use as SetArrayItem(). You must specify the kind of array; if the array exists, it must have the specified form.

`SetStructField()`	Provides access to fields within a nested structure. The namespace for the field is passed as a URI, you need not worry about the path string syntax.
`SetQualifier()`	Provides access to a qualifier attached to a property. The namespace for the qualifier is passed as a URI, you need not worry about the path string syntax.
`SetLocalizedText()`	Modifies the value of a selected item in an alt-text array. Creates an appropriate array item if necessary, and handles special cases for the `x-default` item. See .

These function set property values using binary values:

`SetProperty_Bool()`	Sets the value of a Boolean property from a C++ `bool`.
`SetProperty_Int()`	Sets the value of an integer property from a C `long` integer.
`SetProperty_Int64()`	Sets the value of an integer property from a C `long long` integer.
`SetProperty_Float()`	Sets the value of a floating point property from a C `double float`.
`SetProperty_Date()`	Sets the value of a date/time property from an `XMP_DateTime` struct.

Setting localized text

Setting localized text in alt-text arrays follows these rules:

▶ If the selected item is from a match with the specific language, the value of that item is modified. If the existing value of that item matches the existing value of the `x-default` item, the `x-default` item is also modified. If the array only has one existing item (which is not `x-default`), an `x-default` item is added with the given value.

▶ If the selected item is from a match with the generic language and there are no other generic matches, the value of that item is modified. If the existing value of that item matches the existing value of the `x-default` item, the `x-default` item is also modified. If the array only has one existing item (which is not `x-default`), an `x-default` item is added with the given value.

▶ If the selected item is from a partial match with the generic language and there are other partial matches, a new item is created for the specific language. The `x-default` item is not modified.

▶ If the selected item is from the last two rules then a new item is created for the specific language. If the array only had an `x-default` item, the `x-default` item is also modified. If the array was empty, items are created for the specific language and `x-default`.

Retrieving property values

All of the retrieval functions return a Boolean result telling if the property exists, and, if it does, option flags describing the property. If the property exists and has a value, the function returns the value.

These retrieval functions return values as UTF-8 encoded strings:

`GetProperty()`	The simplest property getter, mainly for top level simple properties or after using the path composition functions in TXMPUtils.
`GetArrayItem()`	Provides access to items within an array. The index is passed as an integer, you need not worry about the path string syntax for array items, convert a loop index to a string, and so on.
`CountArrayItems()`	Useful for iteration within an array.
`GetStructField()`	Provides access to fields within a nested structure. The namespace for the field is passed as a URI, you need not worry about the path string syntax.
`GetQualifier()`	Provides access to a qualifier attached to a property. The namespace for the qualifier is passed as a URI, you need not worry about the path string syntax. **Note:** Currently qualifiers are supported only for simple leaf properties.
`GetLocalizedText()`	Returns information about a selected item in an alt-text array.

These functions return binary property values:

`GetProperty_Bool()`	Returns the value of a Boolean property as a C++ bool.
`GetProperty_Int()`	Returns the value of an integer property as a C long integer.
`GetProperty_Int64()`	Returns the value of an integer property as a C long long integer.
`GetProperty_Float()`	Returns the value of a floating point property as a C double float.
`GetProperty_Date()`	Returns the value of a date/time property as an XMP_DateTime struct.

Deleting and detecting properties

These functions delete properties. It is not an error if the specified property does not exist.

`DeleteProperty()`	Deletes the given XMP subtree rooted at the given property.
`DeleteArrayItem()`	Deletes the given XMP subtree rooted at the given array item.
`DeleteStructField()`	Deletes the given XMP subtree rooted at the given struct field.
`DeleteQualifier()`	Deletes the given XMP subtree rooted at the given qualifier.
`DeleteLocalizedText()`	Deletes a specific language alternative value from an alt-text array.

These functions report whether properties exist:

`DoesPropertyExist()`	Reports whether a property exists.
`DoesArrayItemExist()`	Reports whether an array item exists.

`DoesStructFieldExist()`	Reports whether a struct field exists.
`DoesQualifierExist()`	Reports whether a qualifier exists.

Toolkit configuration

These globally-available static functions, called directly from the `SXMPMeta` class, provide utilities for working with and configuring the XMP Toolkit SDK.

Initialization and termination

These static functions provide a framework for using the Toolkit:

`GetVersionInfo()`	Obtains version information for the XMP Toolkit SDK.
`Initialize()`	Explicitly initializes the XMP Toolkit SDK before use.
`Terminate()`	Terminates the XMP Toolkit SDK.

SXMPIterator class

This class provides iteration services for the `SXMPMeta` object. The `SXMPIterator` functions provide a uniform means to iterate over the schema and properties within an XMP object.

Creating iterator objects

The default constructor creates a new empty `SXMPIterator` object:

```
SXMPIterator();
```

▶ You can pass an existing `SXMPIterator` object to the constructor to copy it. The copy constructor creates a new client iterator that refers to the same underlying iterator.

Pass a specific `SXMPMeta` object to the constructor to create an iterator for the properties in that object. You can specify an iteration root within the property tree, and can provide option flags that control how the iteration is performed.

The options to control the iteration are:

`kXMP_IterJustChildren`	Visit just the immediate children of the root. Skip the root itself and all nodes below the immediate children. This omits the qualifiers of the immediate children, the qualifier nodes being below what they qualify.
`kXMP_IterJustLeafNodes`	Visit just the leaf property nodes and their qualifiers.
`kXMP_IterJustLeafName`	Return just the leaf component of the node names. The default is to return the full path name.

`kXMP_IterIncludeAliases`	Include aliases as part of the iteration. Since aliases are not actual nodes the default iteration does not visit them.
`kXMP_IterOmitQualifiers`	Do not visit the qualifiers of a node.

Performing iterations

Metadata within an XMP object is represented as a data tree with a single root node, which does not explicitly appear in the data dump and is never visited by an iterator. Beneath the root are schema nodes, which collect top-level properties in the same namespace. These are created and destroyed implicitly. Beneath the schema nodes are the property nodes. The nodes below a property node depend on its type (simple, struct, or array) and whether it has qualifiers.

An `SXMPIterator` constructor defines a starting point for the iteration and options that control how it proceeds. By default, iteration starts at the root and visits all nodes beneath it in a depth-first manner. The root node is not visited; the first visited node is a schema node.

You can provide a schema name or property path to select a different starting node. By default, this visits the named root node first, then all nodes beneath it in a depth-first manner.

The `SXMPIterator::Next()` method delivers the schema URI, path, and option flags for the node being visited. For a simple property node, it also delivers the value. Qualifiers for this node are visited next. The fields of a struct or items of an array are visited after the qualifiers of the parent.

The `SXMPIterator` class provides these functions for performing iterations:

`Next()`	Visits the next node in the iteration. Returns true if there was another node to visit, false if the iteration is done.
`Skip()`	Skips some portion of the remaining iterations. You can choose to skip the subtree below the current node, or the subtree below and remaining siblings of the current node.

SXMPUtils class

This class provides utilities that support the use of the main `SXMPMeta` functions.

▶ The [Path composition functions](#) provide support for composing path expressions to deeply nested properties.

▶ The [Type conversion functions](#) provide support for converting property values between binary types and UTF-8 encoded strings.

Path composition functions

The accessor functions in `SXMPMeta` such as `GetProperty()`, `GetArrayItem()`, and `GetStructField()` provide easy access to top-level simple properties, items in top-level arrays, and fields of top-level structs. They do not provide convenient access to more complex things, such as fields several levels deep in a complex struct, fields within an array of structs, or items of an array that is a field of a struct.

You can use the utility functions provided by `SXMPUtils` to construct the paths required to access such deeply nested properties. You can also use them to compose paths to top-level array items or struct fields to use with binary accessors like `GetProperty_Int()`.

You can use these functions to compose a complete path expression, or all but the last component. For example, suppose you have a property that is an array of integers within a struct. You can access one of the array items like this:

```
SXMPUtils::ComposeStructFieldPath( schemaNS, "Struct", fieldNS, "Array", &path );
SXMPUtils::ComposeArrayItemPath( schemaNS, path, index, &path );
exists = xmpObj.GetProperty_Int( schemaNS, path, &value, &options );
```

You could also use this code if you want the string form of the integer:

```
SXMPUtils::ComposeStructFieldPath( schemaNS, "Struct", fieldNS, "Array", &path );
xmpObj.GetArrayItem( schemaNS, path, index, &value, &options );
```

(In these examples, the *schemaNS* value refers to the top-level "schema" namespace, which the XMP Toolkit SDK keeps separate from the rest of the path expression.)

The `SXMPUtils` class defines these path-composition functions:

`ComposeArrayItemPath()`	Composes the path expression for an item in an array.
`ComposeStructFieldPath()`	Composes the path expression for a field in a struct.
`ComposeQualifierPath()`	Composes the path expression for a qualifier.
`ComposeLangSelector()`	Composes the path expression to select an alternate item by language.
`ComposeFieldSelector()`	Composes the path expression to select an alternate item by a field's value.

Composing paths from fields

The path syntax allows two forms of "content addressing" that you can use to select an item in an array of alternatives. The form used in `ComposeFieldSelector()` lets you select an item in an array of structs based on the value of one of the fields in the structs. The other form of content addressing is shown in `ComposeLangSelector()`.

For example, consider a simple struct that has two fields, the name of a city and the URI of an FTP site in that city. Use `ComposeFieldSelector()` to create an array of download alternatives. You can show the user a popup built from the values of the city fields. You can then get the corresponding URI as follows:

```
ComposeFieldSelector( schemaNS, "Downloads", fieldNS, "City", chosenCity, &path );
exists = GetStructField( schemaNS, path, fieldNS, "URI", &uri );
```

Type conversion functions

These functions support the conversion of property values between binary types and UTF-8 encoded strings:

`ConvertFromBool()`	Converts from Boolean to string.
`ConvertFromInt()`	Converts from integer to string.
`ConvertFromFloat()`	Converts from floating point to string.
`ConvertFromDate()`	Converts from date/time to string.

ConvertToBool()	Converts from string to Boolean.
ConvertToInt()	Converts from string to integer.
ConvertToInt64()	Converts from string to 64 bit integer.
ConvertToFloat()	Converts from string to floating point.
ConvertToDate()	Converts from string to date/time.

These functions provide date/time manipulation:

CurrentDateTime()	Obtains the current date and time.
SetTimeZone()	Sets the local time zone.
ConvertToUTCTime()	Guarantees that a time is UTC.
ConvertToLocalTime()	Guarantees that a time is local.
CompareDateTime()	Compares the order of two date/time values.

These functions provide Base64 encoding and decoding:

EncodeToBase64()	Converts from raw data to Base64 encoded string.
DecodeFromBase64()	Decodes from Base64 encoded string to raw data.

3   The XMPFiles Component

This chapter introduces the XMPFiles component of the XMP Toolkit SDK, which supports file input and output for XMP, as opposed to the XMPCore API that supports manipulation of the XMP data itself.

▶ introduces the XMPFiles component and how to use it to access XMP metadata stored in files.

▶ summarizes the functions provided by `SXMPFiles`. For reference details, see the online documentation of the template class, `TXMPFiles`, provided with the XMP Toolkit SDK.

Using XMPFiles for metadata I/O

The XMPFiles component of the XMP Toolkit SDK provides convenient access to the main, or document level, XMP for a file in any supported format. Some file types can have additional XMP packets embedded; for example, if a PDF file has several embedded images, each of the images may have its own XMP data. However, the XMPFiles API retrieves only the document-level XMP.

The XMPFiles API is a front end to a set of file handlers for various formats. The file handlers attempt to provide smart, efficient support for all file formats for which the means to embed XMP is defined in the *XMP Specification*, as documented in *Part 3, Storage in Files*. Adding XMP file-handling support for individual formats is an ongoing effort. These formats currently have specific handlers:

Image formats	DNG (Digital Negative)
	JPEG (Joint Photographic Experts Group)
	PNG (Portable Network Graphics)
	TIFF (Tagged Image File Format)
Dynamic media formats	ASF (Windows Media Audio/Video)
	AVI (Audio-Video Interleaved)
	FLV (Flash Video)
	MOV (QuickTime)
	MP3 (MPEG-1 Audio Layer 3)
	MPEG-2
	MPEG-4
	SWF (Flash)
	WAV (Waveform)
	WMA (Windows Media Audio)
	WMV (Windows Media Video)
Video package formats	AVCHD
	P2
	SonyHDV
	XDCAM
	XDCAM-EX

Adobe application formats	INDD, INDT (Adobe InDesign®) PSD , PSB (Adobe Photoshop)
Document formats	PS, EPS (PostScript® and Encapsulated PostScript) UCF (Universal Container Format)

You can still access metadata in file formats for which there is no specific handler; XMPFiles provides a generic packet-scanning mechanism for files of any format, but its use is not recommended. In particular, you should not use packet-scanning to look for XMP metadata in a file that is not known to have document-level metadata. A Power Point file, for example, could have embedded metadata for an included image, but no document-level metadata. Packet scanning would find the embedded metadata and treat it as document metadata, with unpredictable results.

Where possible, the XMP file handlers allow:

▶ injection of XMP where none currently exists;

▶ expansion of XMP without regard to existing padding;

▶ reconciliation of the XMP and other legacy forms of metadata.

The XMPFiles API is designed for use by clients interested in the metadata and not in the primary file content; the Adobe Bridge application is a typical example. The API is not intended to be particularly appropriate for files authored by an application; that is, those files for which the application has explicit knowledge of the file format.

The XMPFiles API is defined in the template class `TXMPFiles`. Use of the concrete class `SXMPFiles` (based on `std::string`) is standard.

▶ The class defines functions for opening and closing files, and for accessing the embedded metadata, in addition to initialization and informational functions.

▶ The `SXMPFiles` class provides static functions to initialize and terminate the XMPFiles component, retrieve version information for it, and allow you to query the features available in individual format handlers.

For a summary of API functions, see .

Initializing and terminating XMPFiles

You must initialize the XMPFiles component of the Toolkit before you can create instances of the `SXMPFiles` class. Do this by calling the static function `Initialize()` of the concrete class `SXMPFiles`:

```
SXMPFiles::Initialize();
```

The function returns true if the component has been initialized successfully; you can then use the constructor to create `SXMPFiles` instance objects with which to access files and their contained XMP metadata.

Once all processing is complete, you should explicitly terminate both components of the Toolkit, in order to deallocate any global structures that were created on initialization. If you have initialized `SXMPFiles`, terminate it explicitly before terminating `SXMPMeta`, which terminates the entire Toolkit:

```
SXMPMeta::Initialize();
SXMPFiles::Initialize();

// do the metadata work

SXMPFiles::Terminate();
SXMPMeta::Terminate();
```

Accessing metadata in files

Files must be opened before you can read from them. When you construct the `SXMPFiles` object with an explicit file reference and opening options, the constructor also opens the file. You can also create an empty object, however, and use it to open a file explicitly:

```
SXMPFiles myFile;
myFile.OpenFile(filename, file_format, options);
```

Options that you provide with either the constructor or the `OpenFile()` function allow you to open a file for read-only access, or for read-write access. (There are additional details; see <u>"File formats and open options" on page 47</u>).

Once a file has been opened, you can access the metadata. To request the metadata from the `SXMPFiles` instance, use the function `GetXMP()`. This function reports whether the file has metadata, as well as retrieving the metadata if present. The function parses file metadata into an `SXMPMeta` object. Depending on the file format, you can also choose to retrieve the raw packet data.

When calling `GetXMP()`, you must provide at least an `SXMPMeta` instance. The function can take additional arguments to retrieve the raw XMP packet and the packet information. For example:

```
SXMPMeta meta;
std::string packet;
XMP_PacketInfo info;
myFile.GetXMP( &meta, &packet, &info );
```

The call to `GetXMP()` returns true if the file contains XMP data, false if it does not. If it does contain XMP, the function parses the data into the provided `SXMPMeta` instance, which you can then use to access any of the XMP properties, according to how the file was opened.

▶ If you do not need to write out changed metadata, you can specify read-only access when you open the file. In this case, the disk file is automatically closed in the file system after the data is read, and you do not need to explicitly close it.

▶ Typically, you will need to open a file, read and write the metadata, then close the file. To do this, you specify read-write access when you open the file.

  ▷ To modify the metadata in memory, use `SXMPFiles::PutXMP()`. You can do this as often as necessary while the file is open.

  ▷ When you close the file, `SXMPFiles::CloseFile()` writes out the metadata to the disk file.

When processing is complete, you should close the file explicitly, even if the file has been opened with read-only access. Use the function `CloseFile()` on the `SXMPFiles` instance:

```
myFile.CloseFile();
```

▶ If you opened the file as read-only, the disk file is actually closed in the file system immediately after reading the XMP data, in order to avoid blocking the file. You can use `CloseFile()` to close the object any time after calling `GetXMP()`.

▶ If you opened the file for writing, any changes that you make to the metadata with `PutXMP()` are not actually written to the file until you call `CloseFile()`, which assigns the metadata packet back to the file and closes it in the file system.

File formats and open options

The XMPFiles component provides a set of handlers that understand specific file formats. The file format that you specify on creation or with `OpenFile()` helps the XMPFiles component determine which file handler to use for the file. If you provide a specific expected format, XMPFiles uses it as a hint about what file handler to try first. If you want to open only a file of the given type, set the option flag `kXMPFiles_OpenStrictly` (see "Open options" on page 48). This causes XMPFiles to throw an exception if the format doesn't match, rather than going on to check for other formats.

If you do not know the file format, use the default file format constant, `kXMP_UnknownFile`. In this case (or if the format does not match the specified one and the open-strictly option is not set) XMPFiles chooses a handler based on the actual content of the file. It checks each registered file handler in turn to see if it understands the file format until it finds a suitable handler. It uses the file's extension to choose the first handler to try.

The call to `OpenFile()` returns true if the file was successfully opened. If the file could not be opened successfully but no serious errors occurred, the function returns false. For serious errors, for example if the specified file does not exist, an exception is thrown.

You should not attempt to open files that are not known to contain document-level metadata. Power Point files, for instance, might contain embedded metadata for included images, but no document-level metadata.

Querying the file handler

The static function `SXMPMeta::GetFormatInfo()` allows you to determine the extent to which file handling is supported for a particular file format. Depending on the format, the handler can provide these capabilities:

▶ Inject first-time XMP into an existing file.

▶ Expand XMP or other metadata in an existing file.

▶ Copy one file to another, writing new metadata.

▶ Reconcile data between XMP and legacy metadata formats.

▶ Allow access to just the XMP, ignoring other metadata formats.

▶ Retrieve raw XMP packet information.

Open options

In addition to the simple distinction of opening a file for read-only or read-write, you can combine option bit flags to specify some additional control of the operation. Use the logical OR operator to combine the bit-flag constants:

```
XMP_OptionBits opts = kXMPFiles_OpenForRead | kXMPFiles_OpenUseSmartHandler
```

These options are available:

`kXMPFiles_OpenForRead`	Open for read-only access.
`kXMPFiles_OpenForUpdate`	Open for reading and writing.
`kXMPFiles_OpenOnlyXMP`	Only the XMP is wanted, allows space/time optimizations.
`kXMPFiles_OpenStrictly`	Be strict about locating XMP and reconciling with other forms.
`kXMPFiles_OpenUseSmartHandler`	Require the use of a smart handler.
`kXMPFiles_OpenUsePacketScanning`	Force packet scanning, do not use a smart handler.
`kXMPFiles_OpenLimitedScanning`	Only packet scan files "known" to need scanning.
`kXMPFiles_OpenInBackground`	Set if calling from background thread.

Updating and writing file XMP

If a file is open for update, you can inject new XMP or write modified XMP back to a resource file when you close that file; however, you must first update the XMP associated with the file by passing an XMP object containing the new or modified data to the `SXMPFiles` instance in `PutXMP()`.

If the file did not previously contain XMP, or if the modified XMP is larger than before, you might not actually be able to update the file. It depends on the format and the file handler's capabilities, as well as on how the file was opened. It is a good idea to check first, using `SXMPFiles::CanPutXMP()`:

```
if( myFile.CanPutXMP( meta )){
    myFile.PutXMP( meta );
}
```

`CanPutXMP()` returns true if the file can be updated. For example, if the file was opened as read-only, the call returns false.

Remember that the XMP is not actually written back to the file on disk until you close the file object by calling `CloseFile()`. You can use `PutXMP()` to update the XMP any number of times before closing the file and writing out the data.

Many applications check metadata when opening a file. Leaving a file open can cause conflicts with other applications that might wish to access the file while you are working with the metadata. If you plan to keep a file open for longer than a few minutes, it is better practice to open it for read-only, obtain the metadata, and close the file. When you have finished processing the metadata, you can re-open the file briefly for write access.

API summary: SXMPFiles class

This concrete class provides the API for the Adobe XMP Toolkit SDK's XMPFiles component. The class provides convenient access to the main, or document level, XMP for a file.

The general model for metadata access is to open a file, read and write the metadata, then close the file. While open, portions of the file might be maintained in RAM data structures. Memory usage can vary considerably depending on file format and access options. You can open a file for read-only or read-write access, with typical exclusion for both modes.

File handler configuration

Globally-available static functions, called directly from the `SXMPFiles` class, provide utilities for working with and querying the XMP file handler.

These static functions are provided:

`Initialize()`	You must initialize the file handler before using SXMPFiles.
`Terminate()`	You can terminate the file handler when done using SXMPFiles. This deallocates global data structures created by initialization.
`GetVersionInfo()`	Reports version information for the XMP file handler.
`GetFormatInfo()`	Reports the supported features for a given file format. The supported features can vary considerably among file formats, depending on both the general capabilities of the format and the implementation of the handler for that format.
`CheckFileFormat()`	Reports the format of a file, as would be determined when attempting to open that file.
`CheckPackageFormat()`	Reports the format of a package, given the name of the top-level folder. Examples of recognized packages include the video formats P2, XDCAM, or Sony HDV. These packages contain collections of clips, stored as multiple files in specific subfolders.
	This is not the same path you would pass `OpenFile()`. For example, the top-level path for a package might be "…/MyMovie", while the path to a file you wish to open would be "…/MyMovie/SomeClip".

Creating file objects

The default constructor initializes an object that is associated with no file.

```
SXMPFiles();
```

You can pass a file with which to initialize the object; this opens the specified file. The destructor automatically closes the object's associated file if necessary.

You can pass an existing file object to the constructor to copy it; this increments an internal reference count, but does not perform a deep copy; see <u>"Copying metadata" on page 34</u>.

Performing file operations

The functions defined in `SXMPFiles` allow you to open and close files, and retrieve file information. File operation functions include:

`OpenFile()`	Opens a file for metadata access. You can open it for reading, or for updating. If you open a file for update, you must explicitly close it. Options determine whether to reconcile other forms of metadata.
`CloseFile()`	Closes an opened file.
`GetFileInfo()`	Retrieves basic information about an opened file.
`SetAbortProc()`	Allows you to define a callback function that checks for and responds to a user-signaled abort.

Accessing metadata in files

`GetXMP()`	Obtains the parsed XMP from an open file. You can also choose to get the raw XMP packet, and information about the raw XMP packet. Can report whether XMP is present or not.
`PutXMP()`	Updates the XMP in a file. This function supplies new XMP for the file; however, the file is not actually written until closed. The options provided when the file was opened determine whether the function reconciles other forms of metadata.
`CanPutXMP()`	Reports whether the XMP can be updated in the file, for a given XMP packet size.

4   Using the XMP Toolkit SDK

This chapter provides hands-on examples and advice for using the XMP Toolkit to perform typical metadata handling. The examples are in C++, but the usage model is very similar for Java and JavaScript.

▶ "Getting started" on page 51 describes how to build the XMP Toolkit from the projects included with the SDK, and demonstrates how to build your own project using the XMP Toolkit libraries.

▶ "Obtaining and creating XMP data" on page 53 discusses how to use the XMPFiles and XMPCore components together to create, obtain, modify, and store XMP data.

▶ Tutorial examples show how to work with the SDK. All projects and code to accompany the tutorials is included in the SDK, in `<xmpsdk>/samples`; see "Sample code and tools" on page 9.

    ▷ "Walkthrough 1: Opening files and reading XMP" on page 58 shows how to create and configure a project in Windows or in Mac OS. It then demonstrates the basic use of the XMPFiles and XMPCore components, obtaining read-only XMP from a file and examining it through the XMP object. It also shows how to do a data-dump of the XMP object to get a more direct view of its contents.

    ▷ "Walkthrough 2: Modifying XMP" on page 69 demonstrates more sophisticated techniques, opening a file for update, and modifying the contained XMP before writing it back to the file. It also shows how to work with XMP in the form of RDF strings.

    ▷ "Walkthrough 3: Working with a custom schema" on page 73 demonstrates how to work with a custom schema that has complex properties. It shows how to access and modify properties with complex paths using the path composition utilities from the XMP API.

Getting started

This section describes how to build the Toolkit and where to locate the resulting libraries.

Before you begin

In order to build the XMP libraries you must download the Expat XML parser. To use the XMPFiles component, you must install the ZLib compression library.

Installing the Expat XML Parser

You must obtain the Expat XML Parser in order to create the XMP libraries. For a full list of files required to install the Expat XML Parser see `readme.txt` in `<xmpsdk>/third-party/expat`.

1. Obtain a copy of the Expat distribution, minimum version 2.0.1. This can be downloaded from:

    http://sourceforge.net/projects/expat

2. Extract the archive and copy the lib folder to `<xmpsdk>/third-party/expat`.

Installing ZLib

You must obtain and install the ZLib compression library in order to use the XMPFiles component with compressed file formats such as UCF.

1. Obtain a copy of the ZLib distribution from:

    http://www.zlib.net/zlib.html

2. Extract the archive and install as directed in the `Usage.txt` file.

Building the XMP libraries

You must compile and link the projects provided by the XMP Toolkit SDK to produce static libraries for use in Windows , Mac OS, or Linux. The following sections give instructions on how to do this for each platform.

Building the Toolkit in Windows with Visual Studio 2008

1. Navigate to `<xmpsdk>\build\vc9`.

2. Open the solution file `XMPToolkit.sln` with MS Visual Studio 2008.

3. From the menu in MS Visual Studio, choose **Build > Build Solution**. This compiles the Toolkit to produce two static libraries, `XMPCoreStaticDebug.lib` and `XMPFilesStaticDebug.lib`.

After a successful build for the 32-bit debug target, the libraries can be found in:

    <xmpsdk>\public\libraries\windows\debug\

Building the Toolkit in Mac OS with XCode 3.x (3 and above)

1. Navigate to `<xmpsdk>/build/xcode3`.

2. Open the project file `XMPToolkit.xcodeproj` with XCode.

3. From the menu in XCode, choose **Build > Build**. This compiles the Toolkit to produce two static libraries, `XMPCoreStaticDebug.a` and `XMPFilesStaticDebug.a`.

After a successful build for the 32-bit debug target, the libraries can be found in:

    <xmpsdk>\public\libraries\macintosh\debug\

Building the Toolkit in Linux

1. Navigate to `<xmpsdk>/build/gcc4` in a command shell.

2. To build the entire toolkit with a 32-bit debug target, execute this statement:

    ```
    make -f XMPToolkit.mak os=i80386linux stage=debug arch=x86
         libpath=/usr/lib/gcc/i386-redhat-linux/4.1.2 build
    ```

    ▷ You may need to adjust the path for your distribution and installation.

    ▷ You can choose the `clean` or `rebuild` target instead of `build`.

3. To rebuild only the XMPCore component with a 32-bit debug target, execute this statement:

```
make -f XMPCore.mak os=i80386linux stage=debug arch=x86
     libpath=/usr/lib/gcc/i386-redhat-linux/4.1.2 rebuild
```

Obtaining and creating XMP data

This section discusses how to use XMPFiles together with XMPCore to obtain or create new sets of XMP metadata. You can:

▶ Parse one or more buffers containing RDF XML into an XMP object, from which you can manipulate it or write it to a file.

▶ Combine multiple XMP objects into a single set of data.

▶ Serialize the XMP contained in an XMP object into a buffer containing an XMP packet as RDF XML.

▶ Use RDF XML to create an XMP object and then apply the XMP to a file.

Parsing XMP

You can construct an `SXMPMeta` object directly from a buffer. A constructor is available that can accept a single RDF buffer and create the XMP object. The buffer must contain a complete and valid RDF stream, comprising an `rdf:RDF` element containing one or more `rdf:Description` elements.

For example, the following shows some valid RDF that represents several properties from the Dublin Core schema. This RDF can be parsed and used to construct an XMP object.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
        <dc:title>
            <rdf:Alt>
                <rdf:li xml:lang="x-default">An English Title</rdf:li>
                <rdf:li xml:lang="en-US">An English Title</rdf:li>
                <rdf:li xml:lang="fr-FR">Un Titre Francais</rdf:li>
            </rdf:Alt>
        </dc:title>

        <dc:description>
            <rdf:Alt>
                <rdf:li xml:lang="x-default">Green Bush</rdf:li>
            </rdf:Alt>

        <dc:format>image/jpeg</dc:format>
        <dc:creator>
            <rdf:Seq>
                <rdf:li>Author Name</rdf:li>
            </rdf:Seq>
        </dc:creator>
    </rdf:Description>
</rdf:RDF>
```

To parse this into an XMP object, you must supply the RDF as an `XMP_StringPtr`, together with the length of the buffer:

```
string buffer = ""
// populate buffer with valid RDF
SXMPMeta meta( buffer.c_str(), strlen(buffer.c_str()) );
```

The constructor can only parse a single valid RDF stream. If you have multiple buffers, you can either combine them into a single buffer before constructing the object, or you can create an empty object, then use `SXMPMeta::ParseFromBuffer()` to load multiple buffers of data. Each buffer can be any length and does not have to end at a valid XML token or Unicode character. However, all buffers should ultimately produce valid RDF.

The parsing function takes an options flag that specifies whether there are more input buffers to be parsed. Use the option `kXMP_ParseMoreBuffers` if there are more buffers to process:

```
meta.ParseFromBuffer( buffer, buffersize, kXMP_ParseMoreBuffers );
```

When all buffers are processed, terminate input to the XMP object by calling the function again with `kXMP_NoOptions` or 0 (which is the default). The following extract demonstrates using multiple buffers to construct an XMP object:

```
// buffers created here
vector <string>::iterator i;
for(i = buffs->begin();i != buffs->end(); ++i){
    meta.ParseFromBuffer( (*i).c_str(),strlen((*i).c_str()),kXMP_ParseMoreBuffers );
}
meta.ParseFromBuffer( NULL, NULL );
```

The last call to `ParseFromBuffer()` uses null as the input buffer, 0 as the length and the default option flag, to signify that there are no more buffers for input.

Combining XMP objects

You can combine the properties from two XMP objects, creating an updated XMP object with properties from both original objects. Use the utility function `SXMPUtils::ApplyTemplate()` to append the properties from one object to another. Supply a source XMP object and a destination XMP object; the destination object is modified by adding the source properties. The source object remains unmodified.

Supply option flags for `ApplyTemplate()` to determine how the function should handle properties that appear in both objects. You can choose to update, modify, or delete properties in the source.

These option flags are available:

```
kXMPTemplate_ClearUnnamedProperties
kXMPTemplate_AddNewProperties
kXMPTemplate_ReplaceExistingProperties
kXMPTemplate_ReplaceWithDeleteEmpty
kXMPTemplate_IncludeInternalProperties
```

For example the following affects all properties, both internal and external, and adds new properties from the source to the destination:

```
SXMPUtils::ApplyTemplate( &destinationXMP, sourceXMP,
    ( kXMPTemplateAddNewProperties | kXMPTemplate_IncludeInternalProperties ));
```

This adds properties from the source only if they do not already exist in the destination. Arrays and structures are merged. If a property exists in the destination and not in the source, it remains unmodified.

For example, suppose we have two XMP objects, represented by the RDF/XML shown here:

Source RDF/XML

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
        <dc:creator>
            <rdf:Seq>
                <rdf:li>C Name</rdf:li>
            </rdf:Seq>
        </dc:creator>

        <dc:title>
            <rdf:Alt>
                <rdf:li xml:lang="x-default">English US</rdf:li>
            </rdf:Alt>
        </dc:title>

        <dc:subject>
            <rdf:Bag>
                <rdf:li>One</rdf:li>
            </rdf:Bag>
        </dc:subject>

    </rdf:Description>
</rdf:RDF>
```

Destination RDF/XML

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
        <dc:creator>
            <rdf:Seq>
                <rdf:li>A Name</rdf:li>
                <rdf:li>B Name</rdf:li>
            </rdf:Seq>
        </dc:creator>

        <dc:format>image/jpeg</dc:format>

        <dc:subject>
            <rdf:Bag>
                <rdf:li>Two</rdf:li>
                <rdf:li>Three</rdf:li>
            </rdf:Bag>
        </dc:subject>

    </rdf:Description>
</rdf:RDF>
```

If you append properties from the source onto the destination using the given options, the result in the destination object is as follows:

Combined RDF/XML

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
        <dc:creator>
            <rdf:Seq>
                <rdf:li>A Name</rdf:li>
                <rdf:li>B Name</rdf:li>
                <rdf:li>C Name</rdf:li>
            </rdf:Seq>
        </dc:creator>
```

```
            <dc:format>image/jpeg</dc:format>

            <dc:title>
                <rdf:Alt>
                    <rdf:li xml:lang="x-default">English US</rdf:li>
                </rdf:Alt>
            </dc:title>

            <dc:subject>
                <rdf:Bag>
                    <rdf:li>One</rdf:li>
                    <rdf:li>Two</rdf:li>
                    <rdf:li>Three</rdf:li>
                </rdf:Bag>
            </dc:subject>

        </rdf:Description>
</rdf:RDF>
```

In this example:

▶ Both the 'creator' property and the 'subject' property are present in both the source and destination, so the arrays have been merged.

▶ The 'title' property exists in the source, but does not exist in the destination, so it has been added.

▶ The source does not have the 'format' property at all, so it remains unmodified in the destination.

You can use `ApplyTemplate()` to apply your own XMP templates to resources by providing a source XMP object with the template properties.

NOTE: The `SXMPUtils::ApplyTemplate()` function replaces the `SXMPUtils::AppendProperties()` function used for this purpose in previous releases.

Serializing XMP

You can use `SXMPMeta::SerializeToBuffer()` to create a serialized string of XML in valid RDF format from the XMP data contained in the XMP object. The *XMP Specification Part 1, Data and Serialization Models* provides a complete overview of the storage model, describes how XMP is serialized and discusses issues with RDF.

You can provide option flags that control the final format of the serialized RDF. For example, it can be a complete XMP packet (which is the default), or you can specify `kXMP_OmitPacketWrapper`. If you want the XMP data to use as little space as possible, use `kXMP_UseCompactFormat` to produce RDF with a very compact syntax. If you have special formatting requirements, you can specify which newline characters to use or how much indentation to apply to the RDF.

This simple call produces serialized XMP in standard format, with the packet wrapper. It uses the default values for indentation, newline characters and the base indent.

```
string xmpBuffer;
meta.SerializeToBuffer( &xmpBuffer, NULL );
```

The following shows the resulting RDF:

```
<?xpacket begin="_" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="XMP Core 4.1.1">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
```

```
        <rdf:Description rdf:about=""
                xmlns:xmp="http://ns.adobe.com/xap/1.0/">
            <xmp:ModifyDate>2007-07-16T11:33:20+01:00</xmp:ModifyDate>
            <xmp:CreateDate>2007-07-16T11:33:20+01:00</xmp:CreateDate>
            <xmp:MetadataDate>2007-07-24T12:26:06+01:00</xmp:MetadataDate>
        </rdf:Description>

        <rdf:Description rdf:about=""
                xmlns:dc="http://purl.org/dc/elements/1.1/">
            <dc:title>
                <rdf:Alt>
                    <rdf:li xml:lang="x-default">English US</rdf:li>
                    <rdf:li xml:lang="en-US">English US</rdf:li>
                    <rdf:li xml:lang="en-GB">English UK</rdf:li>
                </rdf:Alt>
            </dc:title>

            <dc:description>
                <rdf:Alt>
                    <rdf:li xml:lang="x-default">Green Bush</rdf:li>
                </rdf:Alt>
            </dc:description>

            <dc:format>image/jpeg</dc:format>

        </rdf:Description>
    </rdf:RDF>
</x:xmpmeta>

<?xpacket end="w"?>
```

In contrast, the following code segment serializes XMP data without the XMP packet wrapper, and produces serialized data in compact format:

```
string xmpBuffer;
XMP_Options outOpts = kXMP_OmitPacketWrapper | kXMP_UseCompactFormat;
meta.SerializeToBuffer( &xmpBuffer, outOpts, NULL, "", "", NULL );
```

The following figure shows what the XMP looks like after it has been written to a file:

```
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="XMP Core 4.1.1">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about=""
        xmlns:xmp="http://ns.adobe.com/xap/1.0/"
        xmlns:dc="http://purl.org/dc/elements/1.1/"
       xmp:ModifyDate="2007-07-16T11:33:20+01:00"
       xmp:CreateDate="2007-07-16T11:33:20+01:00"
       xmp:MetadataDate="2007-07-24T12:26:06+01:00"
       dc:format="image/jpeg">

        <dc:title>
            <rdf:Alt>
                <rdf:li xml:lang="x-default">English US</rdf:li>
                <rdf:li xml:lang="en-US">English US</rdf:li>
                <rdf:li xml:lang="en-GB">English UK</rdf:li>
            </rdf:Alt>
        </dc:title>
```

```
        <dc:description>
            <rdf:Alt>
                <rdf:li xml:lang="x-default">Green Bush</rdf:li>
            </rdf:Alt>
        </dc:description>

    </rdf:Description>
  </rdf:RDF>
</x:xmpmeta>
```

Walkthrough 1: Opening files and reading XMP

This tutorial shows how to use the XMPFiles component to open a file and access the XMP data, then use the XMPCore component to read several properties from standard schemas. As an aid to development, it also shows how to create dumps of XMP objects. The tutorial creates the `MyReadingXMP` console application, which shows how to access different properties.

This walkthrough is based on the sample *<xmpsdk>*`/samples/source/ReadingXMP`; see "Sample code and tools" on page 9.

▶ The first section of the walkthrough guides you through creating a project with MS Visual Studio 2008 or XCode 3.x.

For an example of how to create a project in Linux, refer to the makefile that is provided with the samples in *<xmpsdk>*`/samples/build/gcc4`

▶ The second part of the walkthrough guides you through adding the necessary code to open a file and read XMP properties.

Setting up a project

This section guides you through creating a console application which will be used to open a file and read XMP properties.

▶ The SDK provides a set of samples that illustrate coding techniques for various tasks. In addition to the source code for each sample, there is a project file for use with a platform-specific IDE.

  ▷ In Windows, project files are for MS Visual Studio 2008, in the folder *<xmpsdk>*`\samples\build\vc9`.

  ▷ In Mac OS, project files are for XCode 3.x, in the folder *<xmpsdk>*`/samples/build/xcode3`

The sections below provide a complete set of instructions for creating a new project on each of the platforms.

▶ Before you begin creating the project, make sure you have built the necessary libraries. Verify that the library folder exists:

 ▷ In Windows: *<xmpsdk>*`\public\libraries\windows\debug`

  ▷ in Mac OS: *<xmpsdk>*`/public/libraries/macintosh/debug/`

If not, check that you have completed all of the steps in "Getting started" on page 51.

Creating a project with MS Visual Studio

If you are working in Windows, follow these steps:

1. Open MS Visual Studio and choose **File > New Project** to display the New Project Wizard.

2. Select **Win32 Console Application**. Name your project 'MyReadingXMP' and select the location for the project to be saved in `<xmpsdk>\samples\build\vc9`. Deselect **Create directory for solution**.
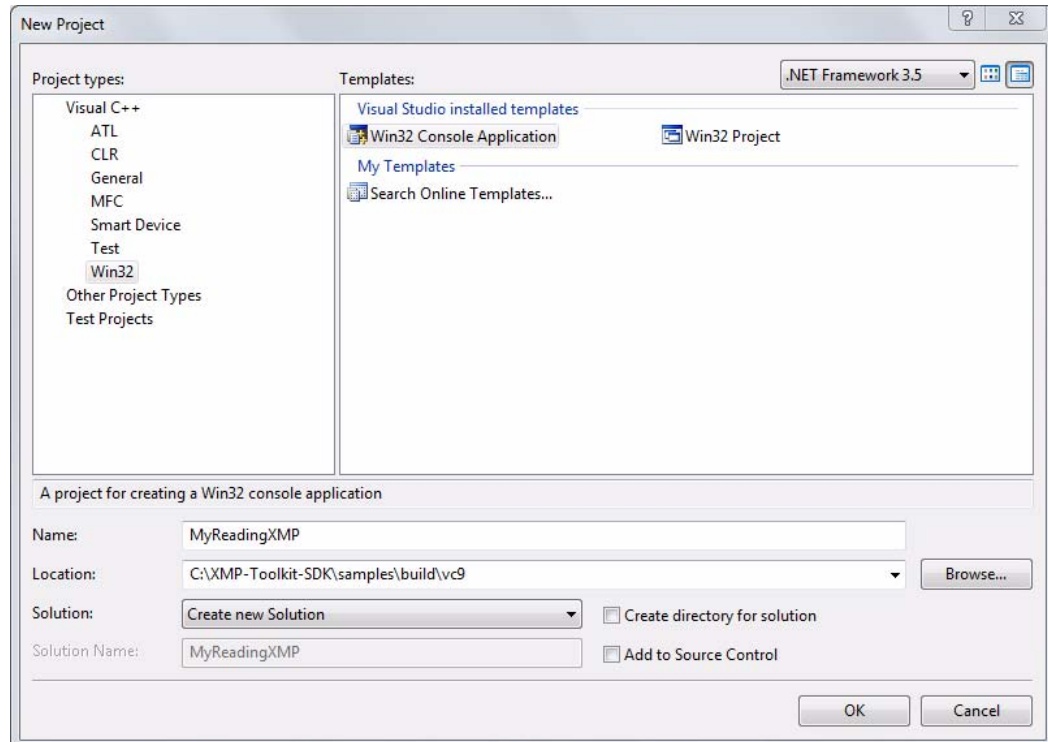

3. Click **OK**. The Win32 Application Wizard appears.

4. Click **Next** on the welcome screen to display the Application Settings.

5. In the Application Settings screen, select **Console Application** for the application type. Select **Empty project**.
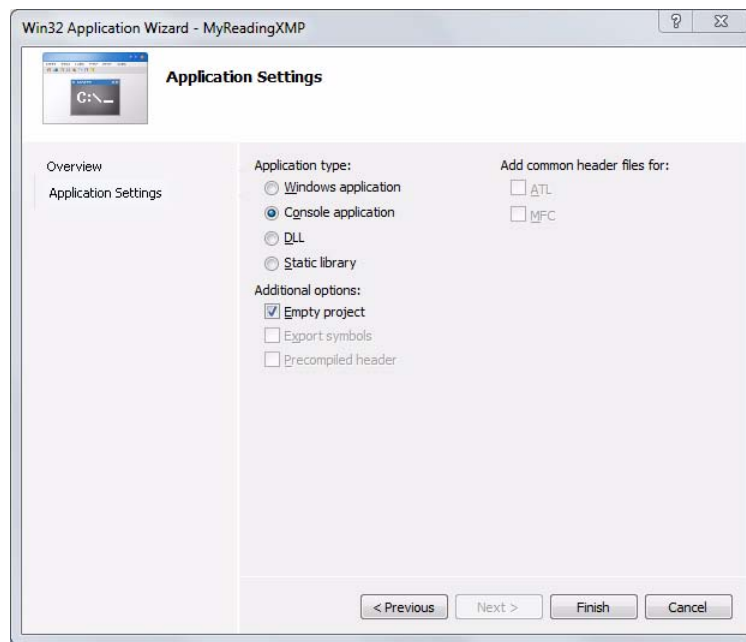
6. Click **Finish**. Your new, empty project is displayed in the Solution Explorer. (If it isn't, choose **View > Solution Explorer**).

7. Select the new project, "MyReadingXMP" in the Solution Explorer, and choose **Project > Add New Item**.

    This displays the Templates that are installed in Visual Studio.
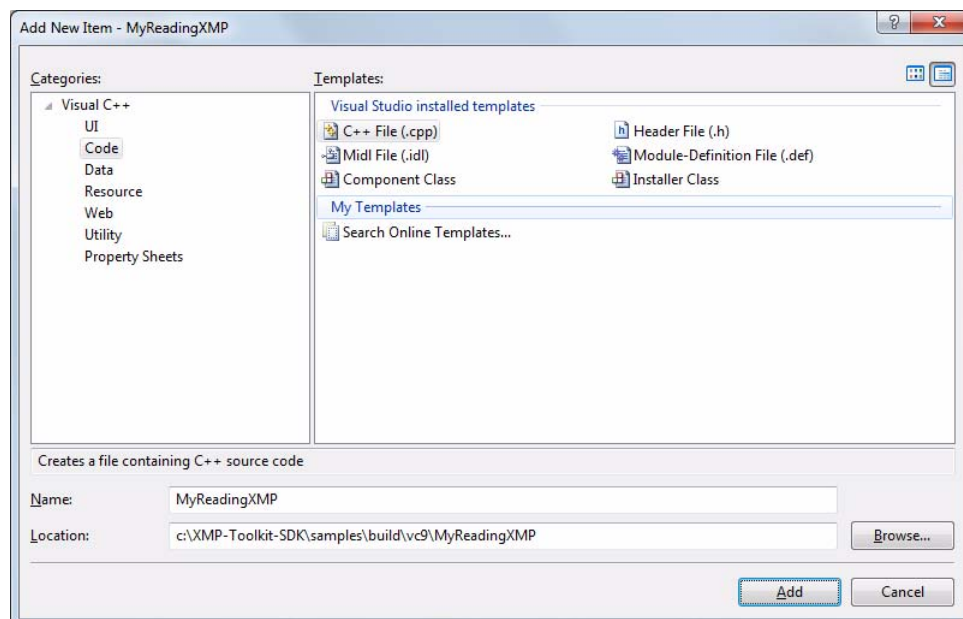
8. Select the template **C++ File (.cpp)** and enter the name 'MyReadingXMP'.

9. Click **Add**.

The basic project is now set up.

Configuring the project

The project's settings must be correct for compiling and linking to the XMPCore and XMPFiles libraries. For the tutorial project, you can use the default configuration for all settings, which is 'Debug'. These project settings are the same for all of the SDK tutorials.

To set the compilation and linking configuration for the project, use these steps:

1. With the project 'MyReadingXMP' selected in the Solution Explorer, choose **Project > Properties**. This displays the Property Pages for the project.

2. Choose **Configuration Properties > C/C++ > General** and set **Additional Include Directories** to point to the *<xmpsdk>*`/public/include` folder.

   ▷ You can use a relative path if you have followed all steps as described; enter `../../../public/include`.

3. Choose **Configuration Properties > C/C++ > Preprocessor** and add the preprocessor definition `WIN_ENV=1`.

4. Select **Configuration Properties > C/C++ > Code Generation** and set **Runtime Library** to **Multi-threaded Debug (/MTd)**.

5. Select **Configuration Properties > Linker > General** and set **Additional Library Directories** to *<xmpsdk>*`/public/libraries/windows/debug`.

   ▷ You can use a relative path if you have followed all steps as described; enter `../../../public/libraries/windows/debug`.

6. Select **Configuration Properties > Linker > Input** and enter `XMPCoreStaticDebug.lib` and `XMPFilesStaticDebug.lib` into the **Additional Dependences** setting. These are the names of the XMP libraries that were created in <u>"Getting started" on page 51</u>
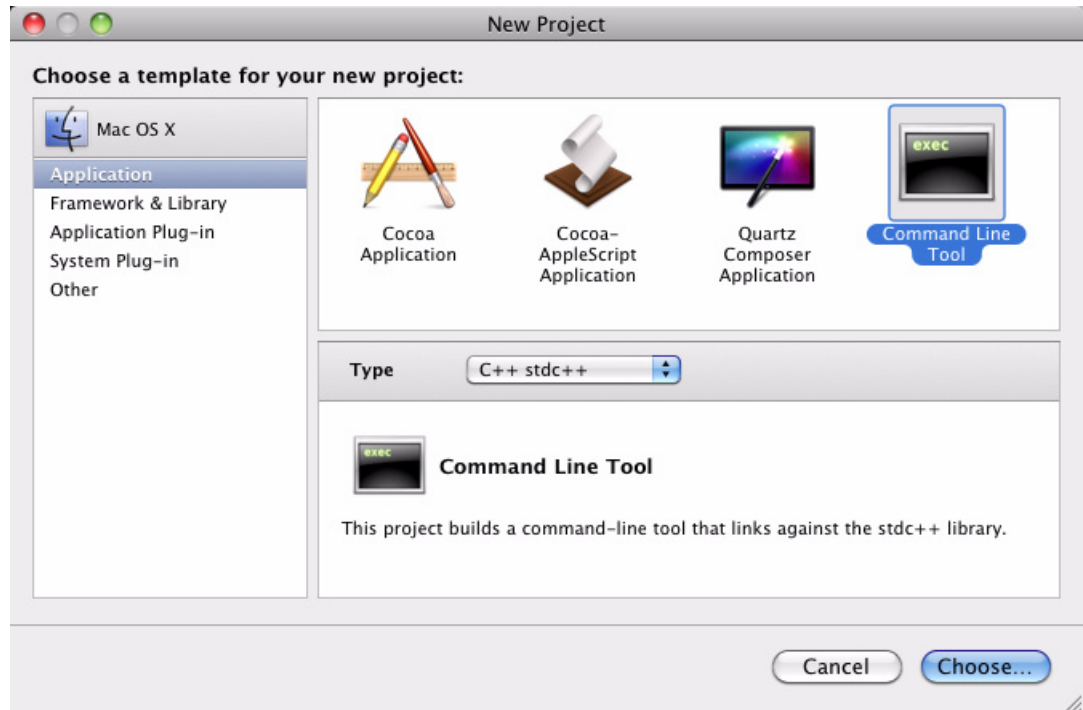
7. Click **Apply**, then **OK**.

The project is now ready for you to add the source code.

Creating a Project with XCode

If you are working in Mac OS, follow these steps:

1. Open XCode 3.2.1 and choose **File > New Project**.

    The New Project dialog appears.

2. Select the **Command Line tool**, and choose the Type "C++ stdc++".

3. Click Choose.

4. Enter the name of the project as 'MyReadingXMP' and select the location for your project as
    `<xmpsdk>/samples/build/xcode3/`.

5. Click **Save** to display your new project.

6. In the project window, highlight the file `main.cpp` and delete it by choosing **Edit > Delete**. In the
    confirmation dialog, select **Also Move to Trash**.

7. In the **Groups and Files** section of the project window, select the **Source** group and choose **File >
    New File**. This displays the **New File Assistant**.

8.   In the "C and C++" section, select **C++ File** and click **Next**.
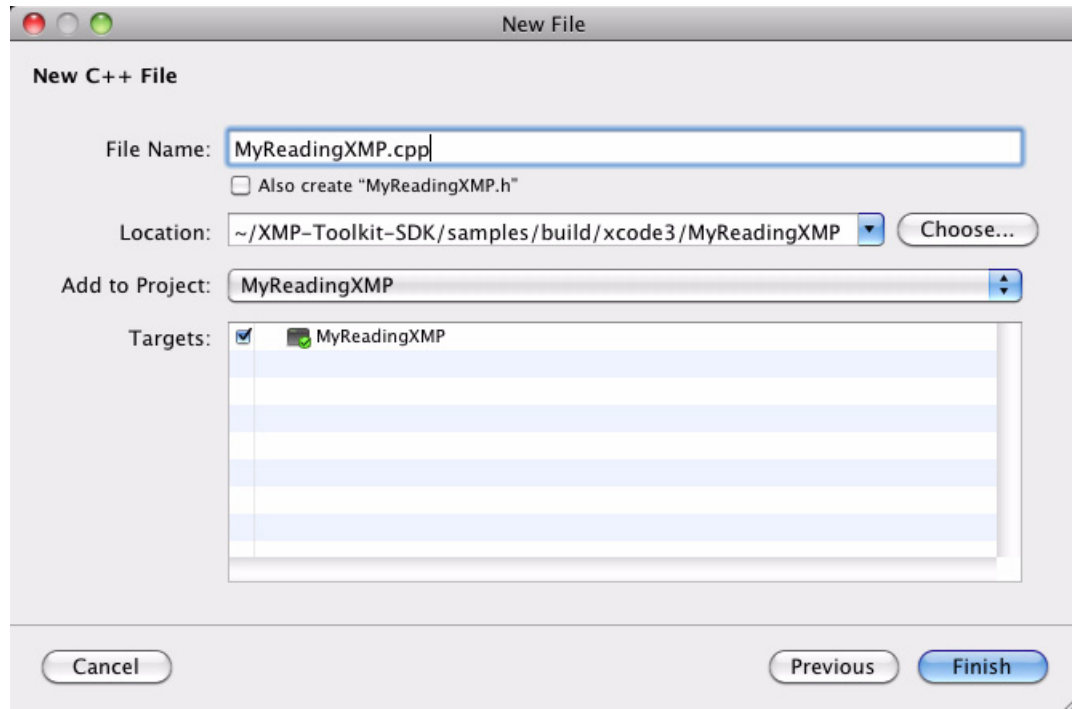
9.   Enter the name for the new file as 'MyReadingXMP.cpp', and deselect the option to create a header file.

10. Click **Finish**.

The basic project is now set up.

Configuring the project

The project's settings must be correct for compiling and linking to the XMPCore and XMPFiles libraries. For the tutorial project, you can use the default configuration for all settings, which is 'Debug'. These project settings are the same for all of the SDK tutorials.

To set the compilation and linking configuration for the project, use these steps:

1.   With the new project selected in the project window, choose **File > Get Info**. This displays the property settings for the project.

2.   Open the Build sections and select **Per-configuration Build Products Path**.

3.   Insert `$(PROJECT_DIR)`, using a space as a separator.

4.   Select **Header Search Paths** and enter the path `${SDKROOT}/public/include`.

5.   Select **Framework Search Paths** and enter `${SDKROOT}/System/Library/Frameworks`.

6.   Deselect **Precompiled Header Uses Files From Build Directory**.

7.   Select **Other Linker Flags** and add the paths to both the XMP Toolkit SDK static libraries:
     `${SDKROOT}/public/libraries/macintosh/debug/libXMPCoreStaticDebug.a`
     `${SDKROOT}/public/libraries/macintosh/debug/libXMPFilesStaticDebug.a`

8. Still in **Other Linker Flags**, at the end of the paths to the static libraries enter the flag `-framework CoreServices`. Enter flag just once, at the end of the second path.

9. Select **Preprocessor Macros** and enter `MAC_ENV=1`.

10. Close the Project Info window.

11. Open the `targets/` folder in the Project window, and select the single build target.

12. Choose **File > Get Info** to display the property settings for the target

13. Select **Preprocessor Macros** and enter `MAC_ENV=1`.

14. Close the Target Info window.

The project is now ready for you to add the source code.

Creating the MyReadXMP application

The application your are about to create will accept a file path to a resource and open the file as read-only, then read the XMP data from the file. Once the XMP packet is available, it will access several properties, display the values in the console window, and also write the results to a file named `XMPDump.txt`. For purposes of illustration, you can observe how these properties describe the resource files that accompany these tutorials, located in the folder `<xmpsdk>/samples/testfiles/`.

The application reads properties from three different schemas, the XMP Basic schema, the Dublin Core schema and the EXIF schema. The properties demonstrate a variety of property and value types:

Schema	Properties	Description
XMP Basic	CreatorTool	A simple property which stores the name of the first known tool used to create the resource.
	MetadataDate	A date-time value that represents the last time any metadata for the resource was changed.
Dublin Core	creator	An ordered array that holds the authors of the resource.
	title	A language alternative that stores the name given to the resource.
	subject	An unordered array that stores phrases or keywords which specify the content of the resource.
EXIF	Flash	A structure that describes the flash state when a photograph was taken.

The following steps provide the major outline of creating the application; they do not show all code used in the application. For the full code listing see `<xmpsdk>/samples/source/ReadingXMP.cpp`.

1. Provide the string class with which to instantiate the template classes:

```
#include <string>
#define TXMP_STRING_TYPE std::string
```

2. Include the XMPFiles component of the XMP Toolkit:

```
#define XMP_INCLUDE_XMPFILES 1
```

3. Ensure that the XMP class templates are instantiated:

```
#include "XMP.incl_cpp"
```

4. Provide access to the XMP API:

```
#include "XMP.hpp"
```

5. Include streaming support:

```
#include <iostream >
#include <fstream>
using namespace std;
```

6. Create a main method. Inside the main method, initialize both XMPCore and XMPFiles, using conditional statements to verify initialization was successful:

```
if (!SXMPMeta::Initialize()) exit(1);
if (!SXMPFiles::Initialize()) exit(1);
```

7. Create the options to open the file for read-only access and request to use a format-specific handler:

```
XMP_OptionBits opts = kXMPFiles_OpenForRead | kXMPFiles_OpenUseSmartHandler;
```

8. The file to be read is provided as a parameter on the command line. Create an `SXMPFiles` instance and open the file, providing the filename, the file format and the options.

   Depending on the file type, there may be no appropriate handler available. In this case, you would have to open the file using packet scanning, providing a different set of option bits:

```
std::string status = "";
SXMPFiles myfile;

    // First, try to open the file
    bool ok = myFile.OpenFile(filename, kXMP_UnknownFile, opts);
    if( ! ok ){
        status += "No smart handler available for " + filename + "\n";
        status += "Trying packet scanning.\n";

        // Now try using packet scanning
        opts = kXMPFiles_OpenForUpdate | kXMPFiles_OpenUsePacketScanning;
        ok = myFile.OpenFile(filename, kXMP_UnknownFile, opts);
    }
```

9. Create the `SXMPMeta` instance and retrieve the XMP from the file. We are not concerned with the raw packet or the packet information, so you can leave out those output parameters:

```
if(ok){
    SXMPMeta meta;
    myFile.GetXMP( &meta );
}
```

10. Within the same logic block as the created `SXMPMeta` instance, add the code to display the simple property "CreatorTool" by providing the namespace URI, the name of the property and a pointer to a string in which to return the property value:

```
bool exists;
string simpleValue;
exists = meta.GetProperty( kXMP_NS_XMP, "CreatorTool", &simpleValue, NULL );
if( exists )
    cout << "CreatorTool = " << simpleValue << endl;
else
    simpleValue.clear();
```

This checks that the property exists, so that the variable has been assigned a valid value, before attempting to write that value to the console.

11. Display the first element of the 'creator' array. Provide the namespace URI, the name of the array, the array index (1 for the first element) and a string in which to return the value:

```
string elementValue;
exists = meta.GetArrayItem( kXMP_NS_DC, "creator", 1, &elementValue, NULL );
if( exists )
    cout << "dc:creator = " << elementValue << endl;
else
    elementValue.clear();
```

12. Traverse the 'subject' property (an array) and display all elements. Use the number of items in the array to control the traversal:

```
string propValue;
int arrSize = meta.CountArrayItems( kXMP_NS_DC, "subject");
for( int i = 1;i <= arrSize;i++ ){
    meta.GetArrayItem( kXMP_NS_DC, "subject", i, &propValue, NULL );
    cout << "dc:subject[" << i << "] = " << propValue << endl;
}
```

13. Get a localized text item; display the 'title' property in English:

```
string itemValue;
meta.GetLocalizedText( kXMP_NS_DC, "title", "en", "en-US", NULL,
    &itemValue, NULL );
cout << "dc:title in English = " << itemValue << endl;
```

14. Get a localized text item; display the 'title' property in French:

```
meta.GetLocalizedText( kXMP_NS_DC, "title", "fr", "fr-FR", NULL,
    &itemValue, NULL );
cout << "dc:title in French = " << itemValue << endl;
```

15. Get a date property; read the 'MetadataDate' property if it exists. If so convert the `XMP_DateTime` into a string and display it:

```
XMP_DateTime myDate;
if( meta.GetProperty_Date( kXMP_NS_XMP, "MetadataDate", &myDate, NULL )){
    string myDateStr;
    SXMPUtils::ConvertFromDate( myDate, &myDateStr );
    cout << "meta:MetadataDate = " << myDateStr << endl;
}
```

16. Discover if the EXIF Flash structure is available, if so display the flash status at the time the photograph was taken:

```
bool exist;
string path, value;
exist = meta.DoesStructFieldExist( kXMP_NS_EXIF, "Flash", kXMP_NS_EXIF,"Fired" );
if( exist ){
    bool flashFired;
    SXMPUtils::ComposeStructFieldPath( kXMP_NS_EXIF, "Flash", kXMP_NS_EXIF,
        "Fired", &path );
    meta.GetProperty_Bool( kXMP_NS_EXIF, path.c_str(), &flashFired, NULL );
    string flash = (flashFired) ? "True" : "False";
    cout << "Flash Used = " << flash << endl;
}
```
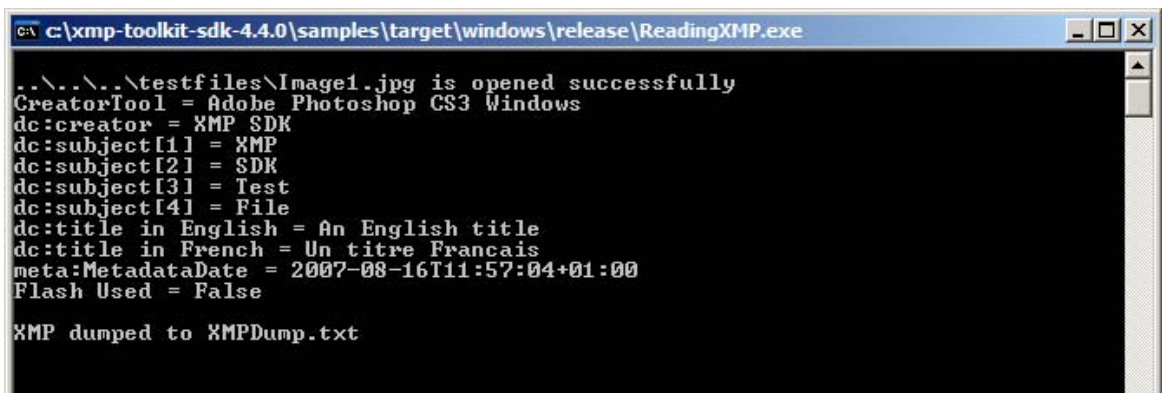
17. Close the `SXMPFiles` instance:

```
myFile.CloseFile();
```

18. Outside of the logic block used to initialize XMPFiles, terminate both XMPCore and XMPFiles.

```
SXMPFiles::Terminate();
SXMPMeta::Terminate();
```

19. If you are going to run the application within your IDE, place a break point on the very last line of the application. If you are going to run it from the command line, this is not necessary.

20. Compile and run the application. You should see output similar to this in the console window:

Adding a debugging callback

For easy viewing of the XMP within a file, you can add a callback function to support dumping of XMP objects to a text file.

1. In `MyReadingXMP.cpp`, add a callback function just above the main function. The function must conform to the signature for the typedef `XMP_TextOutputProc`. Name the function `DumpXMPToFile()`:

```
XMP_Status DumpXMPToFile(void *ref,XMP_StringPtr buff,XMP_StringLen buffSize)
```

2. Inside the function, cast the **ref* to an *ofstream* and direct the buffer to the file. Use the *buffSize* parameter to control how much data is written to the file:

```
ofstream * outFile = static_cast<ofstream*>(ref);
(*outFile).write ( buff, buffSize );
```

3. Ensure the callback returns a valid `XMP_Status`:

```
XMP_Status status = 0;
return status;
```

4. Within the main function add a call to `DumpObject()`. Add this just before the XMPFiles instance is closed:

```
ofstream dumpFile;
dumpFile.open("XMPDump.txt", ios::out);
meta.DumpObject(DumpXMPToFile, &dumpFile);
dumpFile.close();
cout << endl << "XMP dumped to XMPDump.txt" << endl;
```

5. Compile and run the application.

6. Navigate to the appropriate target file of the compiled application and open the file `XMPDump.txt`, which now contains a dump of the XMP data for the file provided as a parameter to the command.

Walkthrough 2: Modifying XMP

This tutorial demonstrates how to modify XMP properties from standard schemas. It shows you how to construct an XMP object from an RDF stream, apply updates to the metadata values, and write the modified XMP back to the resource. It also covers serializing XMP into a string containing an XMP packet in RDF notation.

This walkthrough is based on the sample `<xmpsdk>/samples/source/ModifyingXMP`. Sample resource files accompany these tutorials, located in the folder `<xmpsdk>/samples/testfiles`. See "Sample code and tools" on page 9.

Creating the MyModifyXMP application

Follow the steps in "Setting up a project" on page 58 to set up a project for the console application that will be used for this walkthrough. Name this project `MyModifyingXMP` and add a file named `MyModifyingXMP.cpp`.

The application you are about to create will accept a file path to a resource, open the file for update, and read the XMP data from the file.

You will modify or add several properties to that XMP using XMPCore functions. You will then create an XMP object from an RDF stream and append its properties to the first XMP object. Finally you will serialize the resulting XMP and write it to a file.

Many of the initial steps are essentially the same as those in the previous tutorial; use the directions in "Creating the MyReadXMP application" on page 64.

1. Add the necessary headers and macros to use both components of the XMP Toolkit SDK. See steps 1 to 5 of "Creating the MyReadXMP application" on page 64.

2. Initialize both XMPCore and XMPFiles. See step 6 of "Creating the MyReadXMP application" on page 64.

3. Create the options to open the file for update, using a format-specific handler:

    ```
    XMP_OptionBits opts = kXMPFiles_OpenForUpdate | kXMPFiles_OpenUseSmartHandler;
    ```

4. Create an `SXMPFiles` instance and open the file providing the filename, the file format and the options. See steps 7 to 9 of "Creating the MyReadXMP application" on page 64.

Modifying XMP properties

These steps provide the major outline of creating the application; they do not show all code used in the application. For the full code listing see `<xmpsdk>/samples/source/ModifyingXMP.cpp`.

1. Add code that will write the current values for the properties used in this tutorial to the console window. You can copy the utility function named `displayPropertyValues()`, provided in `<xmpsdk>/samples/source/ModifyingXMP.cpp`. Place the copy in your source file just above the main function.

2. In the main function, just after the call to `GetXMP()`, display the desired property values in the console by calling `displayPropertyValues()`:

    ```
    displayPropertyValues( &meta );
    ```

3. Modify the 'CreatorTool' property. First check the property exists and then apply the edit.

```
if(meta.DoesPropertyExist( kXMP_NS_XMP, "CreatorTool" )){
    meta.SetProperty( kXMP_NS_XMP, "CreatorTool", "Updated By XMP SDK", NULL );
}
```

4. Set the 'MetadataDate' property to the current date and time:

```
XMP_DateTime updatedTime;
SXMPUtils::CurrentDateTime( &updatedTime );
if( meta.DoesPropertyExist( kXMP_NS_XMP, "MetadataDate" )){
    meta.SetProperty_Date( kXMP_NS_XMP, "MetadataDate", updatedTime, NULL );
}
```

5. Create a new 'dc:creator' property, an ordered array. Set the first item of the array to 'Author Name'. Note how you must supply the correct options flag to create an array of a particular type:

```
meta.AppendArrayItem( kXMP_NS_DC, "creator", kXMP_PropArrayIsOrdered,
    "Author Name", NULL );
```

6. Add a new item to the new 'creator' array. Give the item a value of 'Another Author Name':

```
meta.AppendArrayItem( kXMP_NS_DC, "creator",
    kXMP_PropArrayIsOrdered, "Another Author Name", NULL );
```

7. Update the 'dc:title' property. Set two items, one for English and one for French. This also creates the x-default item:

```
meta.SetLocalizedText( kXMP_NS_DC, "title", "en", "en-US", "An English Title" );
meta.SetLocalizedText( kXMP_NS_DC, "title", "fr", "fr-FR", "Un Titre Francais" );
```

8. Add code to write the current values for the properties used in this tutorial to the console. See step 1.

9. Close the `SXMPFiles` instance and terminate XMPFiles and XMPCore, then see steps 17 and 18 of "Creating the MyReadXMP application" on page 64.

10. If you are running the application in an IDE, place a break point on the very last line of the application.

11. Compile and run the application. The output in the console looks like the following. Note, however, that no data has yet been written back to the file.

Using RDF to create XMP

Up to this point the application has been modifying properties by editing them directly. Next you will create an XMP object from RDF and append the newly created object to the existing one.

To demonstrate creating an XMP object from multiple buffers you will create a string of RDF XML. The RDF represents the 'subject' property from the Dublin Core schema. To simulate using multiple buffers, we pass ten characters at a time to the XMP object using `ParseFromBuffer()`.

1. Create a new function just above the main function, that takes no parameters and returns an `SXMPMeta` instance. Name the function `createXMPFromRDF()`. You can copy the code from `<xmpsdk>/samples/source/ModifyingXMP.cpp`.

2. Inside the body of the function create a string to hold the following RDF:

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
  <rdf:Description rdf:about='' xmlns:dc='http://purl.org/dc/elements/1.1/'>
    <dc:subject>
        <rdf:Bag>
            <rdf:li>XMP</rdf:li>
            <rdf:li>SDK</rdf:li>
            <rdf:li>Sample</rdf:li>
        </rdf:Bag>
    </dc:subject>
    <dc:format>image/jpeg</dc:format>
  </rdf:Description>
</rdf:RDF>
```

3. Create an empty XMP object.

4. Loop over the RDF string and pass ten characters at a time to the XMP object using `ParseFromBuffer()`. This is demonstrates how to create an XMP object from multiple buffers. Note the use of the option flag `kXMP_ParseMoreBuffers`:

```
SXMPMeta meta;
int i;
for(i = 0; i < (long)strlen( rdf ) - 10; i += 10)
{
    meta.ParseFromBuffer( &rdf[i], 10, kXMP_ParseMoreBuffers);
}
```

5. Terminate the input to the XMP object by supplying the final piece of RDF. For this final buffer, allow the options flag to default to 0:

```
meta.ParseFromBuffer( &rdf[i], strlen(rdf) - i );
```

6. Return the XMP object from the function.

7. Create a new XMP object by calling the `createXMPFromRDF()` in the main function, just before the call to close the `SXMPFiles` instance:

```
SXMPMeta rdfMeta = createXMPFromRDF();
```

8. Append the properties of the XMP object created with the RDF to the XMP object retrieved from the file. This adds the subject property from the Dublin Core schema:

```
SXMPUtils::ApplyTemplate( &meta, rdfMeta, ( kXMPTemplate_IncludeInternalProperties
    | kXMPTemplate_AddNewProperties | kXMPTemplate_ReplaceExistingProperties ) );
```

9. Add code to write the current values for the properties used in this tutorial to the console. See step 1 from "Modifying XMP properties" on page 69.

10. Place a break point on the last line of the application.

11. Compile and run the application. The output in the console window looks like the following:

```
c:\xmp-toolkit-sdk-4.4.0\samples\target\windows\release\ModifyingXMP.exe

..\..\..\testfiles\Image1.jpg is opened successfully
meta:CreatorTool = Adobe Photoshop CS3 Windows
dc:creator[1] = XMP SDK
dc:creator[2] = XMP SDK
dc:subject[1] = XMP
dc:subject[2] = SDK
dc:subject[3] = Test
dc:subject[4] = File
dc:title in English = An English title
dc:title in French = Un titre Francais
meta:MetadataDate = 2007-08-16T11:57:04+01:00
----------------------------------------
After update:
meta:CreatorTool = Updated By XMP SDK
dc:creator[1] = XMP SDK
dc:creator[2] = Author Name
dc:subject[1] = XMP
dc:subject[2] = SDK
dc:subject[3] = Test
dc:subject[4] = File
dc:title in English = An English title
dc:title in French = Un titre Francais
meta:MetadataDate = 2008-08-14T12:31:39+02:00
----------------------------------------
After Appending Properties:
meta:CreatorTool = Updated By XMP SDK
dc:creator[1] = XMP SDK
dc:creator[2] = Author Name
dc:subject[1] = XMP
dc:subject[2] = SDK
dc:subject[3] = Sample
dc:title in English = An English title
dc:title in French = Un titre Francais
meta:MetadataDate = 2008-08-14T12:31:39+02:00
----------------------------------------
```

Serializing the updated XMP

After all updates have been made to the XMP, we can serialize it to view the changes. These steps walk you through serializing the modified XMP first in standard format and again using a compact RDF format.

1. Create a new function just above the main function, that accepts two arguments, and returns nothing. The first argument is a string pointer that will point to the RDF produced by the serialization. The second argument is a string, the path to a file to which to write the RDF. Name the function `writeRDFToFile()`. The function signature is:

    ```
    void writeRDFToFile( string * rdf, string filename )
    ```

2. Inside the function add code to open a file and write the RDF to it:

    ```
    ofstream outFile;
    outFile.open( filename.c_str(), ios::out );
    outFile << *rdf;
    outFile.close();
    ```

3. Serialize the XMP object, specifying the default values for space and indentation and setting no option flags. Add this code inside the main function, just before the call to close the `SXMPFiles` instance:

    ```
    string xmpBuffer;
    meta.SerializeToBuffer( &xmpBuffer, NULL, NULL, "", "", NULL );
    ```

4. Write the data to a file using your utility function `writeRDFToFile()`:

    ```
    writeRDFToFile(&xmpBuffer, filename+"_XMP_RDF.txt");
    ```

5. Serialize the XMP object again, this time with options to omit the XMP packet wrapper and serialize the RDF in compact format. This call omits the *newline*, *indent*, and *baseIndent* parameters, allowing them to default. Write the data to a different file:

    ```
    XMP_OptionBits outOpts = kXMP_OmitPacketWrapper | kXMP_UseCompactFormat;
    meta.SerializeToBuffer( &xmpBuffer, outOpts );
    writeRDFToFile(&xmpBuffer, filename+"_XMP_RDF_Compact.txt");
    ```

6. Both output files can be found in the same folder as the test file. Open each file and review the contents. Note that the XMP properties are exactly same, although the RDF is different.

Writing the updated XMP back to the file

To this point, all of the changes have been made in the XMP object; no properties have been written back to the resource file. These steps show how to update the `SXMPFiles` instance so that the modifications are written back to the file upon close.

1. Associate the modified XMP with the `SXMPFiles` instance. Place this just before the call to close the `SXMPFiles` instance:

    ```
    if( myFile.CanPutXMP( meta )){
        myFile.PutXMP( meta );
    }
    ```

2. Compile and run the application. The call to `CloseFile()` actually writes out the modified XMP back to the resource file.

Walkthrough 3: Working with a custom schema

This tutorial demonstrates how to work with a custom schema that has complex properties. It shows how to access and modify properties with complex paths using the path composition utilities from the XMP API.

This walkthrough is based on the sample *<xmpsdk>*`/samples/source/CustomSchema`. Sample resource files accompany these tutorials in the folder *<xmpsdk>*`/samples/testfiles`. See <u>"Sample code and tools" on page 9</u>.

In the course of this walkthrough, you will add custom properties to an XMP object, with complex structures and values, demonstrating how to access nested structs, arrays, and language alternatives. You will then use that object to write the metadata to a file.

A schema is provided for the walkthrough and is used to demonstrate how to register you own schema and add complex properties. This schema is purely illustrative, and not intended for actual use. It contains some properties that are available in other schemas, but is not intended to replace those. The sample schema is designed to track document changes; it provides properties to record the authors of a document, along with contact details for each author, and the document changes each author has made.

Creating the MyCustomSchema application

Follow the steps in "Setting up a project" on page 58 to set up a project for the console application that will be used for this walkthrough. Name this project `MyCustomSchema` and add a file named `MyCustomSchema.cpp`.

Many of the initial steps are essentially the same as those in the first tutorial; use the directions in "Creating the MyReadXMP application" on page 64.

1. Add the necessary headers and macros. You do not need to include XMPFiles. See steps 1 to 5 of "Creating the MyReadXMP application" on page 64.

2. Initialize the XMPCore component; see step 6 of "Creating the MyReadXMP application" on page 64. You do not need to initialize XMPFiles.

Creating a custom schema

These steps provide the major outline of creating the application; they do not show all code used in the application. For the full code listing see *<xmpsdk>*/samples/source/CustomSchema.cpp.

1. Register the namespaces and the prefixes that will be used for the schema:

```
const XMP_StringPtr  kXMP_NS_SDK_EDIT = "http://ns.adobe/xmp/sdk/Edit/";
const XMP_StringPtr  kXMP_NS_SDK_USERS = "http://ns.adobe/xmp/sdk/User/";
// ...
string actualPrefix;
SXMPMeta::RegisterNamespace( kXMP_NS_SDK_EDIT, "xsdkEdit", &actualPrefix );
SXMPMeta::RegisterNamespace( kXMP_NS_SDK_USERS, "xsdkUser",&actualPrefix );
```

Creating complex properties

1. Create an XMP object and create the `DocumentUsers` property as an unordered array, using the correct option bits for the array items. The array will store structures, so use `kXMP_PropValueIsStruct`:

```
SXMPMeta meta;
meta.AppendArrayItem( kXMP_NS_SDK_EDIT, "DocumentUsers", kXMP_PropValueIsArray,
     NULL, kXMP_PropValueIsStruct );
```

2. Compose the path to the last item, a `UserDetails` structure, in the `DocumentUsers` array:

```
string userItemPath;
SXMPUtils::ComposeArrayItemPath( kXMP_NS_SDK_EDIT, "DocumentUsers",
   kXMP_ArrayLastItem, &userItemPath );
```

3. We need to add fields and their values to the `UserDetails` structure. First, set the `User` field for the `UserDetails` structure, using the path that was composed in the previous step:

```
meta.SetStructField( kXMP_NS_SDK_EDIT, userItemPath.c_str(),
   kXMP_NS_SDK_USERS, "User", "John Smith" );
```

4. Add a qualifier to the new `User` field. Create a path to the field, then use this path to set the qualifier value:

```
string userFieldPath;
SXMPUtils::ComposeStructFieldPath( kXMP_NS_SDK_EDIT, userItemPath.c_str(),
    kXMP_NS_SDK_USERS, "User", &userFieldPath );
meta.SetQualifier( kXMP_NS_SDK_EDIT, userFieldPath.c_str(),
    kXMP_NS_SDK_USERS, "Role", "Dev Engineer" );
```

5. To set the DUID field, compose a path to the field and set the binary value using `SetProperty_Int()`:

```
string duidPath;
SXMPUtils::ComposeStructFieldPath( kXMP_NS_SDK_EDIT, userItemPath.c_str(),
    kXMP_NS_SDK_USERS, "DUID", &duidPath );
meta.SetProperty_Int( kXMP_NS_SDK_EDIT, duidPath.c_str(), 2 );
```

6. Add the `ContactDetails` field, which is itself a struct of type `Contact`. The options bit `kXMP_PropValueIsStruct` makes the new property a struct:

```
meta.SetStructField( kXMP_NS_SDK_EDIT, userItemPath.c_str(), kXMP_NS_SDK_USERS,
    "ContactDetails", NULL, kXMP_PropValueIsStruct );
```

7. Compose a path to the new structure:

```
string contactStructPath;
SXMPUtils::ComposeStructFieldPath( kXMP_NS_SDK_EDIT, userItemPath.c_str(),
    kXMP_NS_SDK_USERS, "ContactDetails", &contactStructPath );
```

8. Use the composed path to the `ContactDetails` field to add the three fields to the structure; `Email`, `Telephone`, and `BaseLocation`. In each case, we must provide the correct options bit to describe the property:

```
meta.SetStructField( kXMP_NS_SDK_EDIT, contactStructPath.c_str(),
    kXMP_NS_SDK_USERS, "Email", NULL, kXMP_PropArrayIsAlternate );
```

```
meta.SetStructField( kXMP_NS_SDK_EDIT, contactStructPath.c_str(),
    kXMP_NS_SDK_USERS, "Telephone", NULL, kXMP_PropValueIsArray );
```

```
meta.SetStructField( kXMP_NS_SDK_EDIT, contactStructPath.c_str(),
    kXMP_NS_SDK_USERS, "BaseLocation", "", NULL );
```

9. Compose a path to the `Email` field using the previously composed path to the `ContactDetails` field, and add two items to the array:

```
string path;

SXMPUtils::ComposeStructFieldPath( kXMP_NS_SDK_EDIT,
    contactStructPath.c_str(), kXMP_NS_SDK_USERS, "Email", &path );

meta.AppendArrayItem( kXMP_NS_SDK_EDIT, path.c_str(), NULL, "js@adobe.xmp.com" );
meta.AppendArrayItem( kXMP_NS_SDK_EDIT, path.c_str(), NULL, "js@adobe.home.com" );
```

10. Do the same for the `Telephone` field:

```
SXMPUtils::ComposeStructFieldPath( kXMP_NS_SDK_EDIT,
    contactStructPath.c_str(), kXMP_NS_SDK_USERS, "Telephone", &path );

meta.AppendArrayItem( kXMP_NS_SDK_EDIT, path.c_str(), NULL, "89112" );
meta.AppendArrayItem( kXMP_NS_SDK_EDIT, path.c_str(), NULL, "84432" );
```

11. Add the `BaseLocation` field value:

```
SXMPUtils::ComposeStructFieldPath( kXMP_NS_SDK_EDIT,
    contactStructPath.c_str(), kXMP_NS_SDK_USERS, "BaseLocation", &path );

meta.SetProperty( kXMP_NS_SDK_EDIT, path.c_str(), "London" );
```

You have now completed adding the `DocumentUsers` property to the XMP object, with values for one user. Here is the debugging dump of this object:

```
xsdkEdit: http://ns.adobe/xmp/sdk/Edit/ (0x80000000 : schema)
    xsdkEdit:DocumentUsers (0x200 : isArray)
        [1] (0x100 : isStruct)
            xsdkUser:User = "John Smith" (0x10 : hasQual)
                ? xsdkUser:Role = "Dev Engineer" (0x20 : isQual)
            xsdkUser:DUID = "2"
            xsdkUser:ContactDetails (0x100 : isStruct)
                xsdkUser:Email (0xE00 : isAlt isOrdered isArray)
                    [1] = "js@adobe.xmp.com"
                    [2] = "js@adobe.home.com"
                xsdkUser:Telephone (0x200 : isArray)
                    [1] = "89112"
                    [2] = "84432"
                xsdkUser:BaseLocation = "London"
```

Adding more properties

We will now add the second set of properties from the custom schema, to represent editing actions in the described document and note what actions have taken place. For this sample we add data that shows when the document was created.

1. Create a new property for the `DocumentEdit` ordered array, an array that will hold `EditDetail` structures:

```
meta.AppendArrayItem( kXMP_NS_SDK_EDIT, "DocumentEdit",
    kXMP_PropArrayIsOrdered, NULL, kXMP_PropValueIsStruct );
```

2. Compose the path to the last item of the new array. Because we have not created any values yet, this points to an `EditDetails` structure at the first index of the array:

```
string lastItemPath;
SXMPUtils::ComposeArrayItemPath( kXMP_NS_SDK_EDIT, "DocumentEdit",
    kXMP_ArrayLastItem, &lastItemPath );
```

3. Use the composed path to add the `EditDate` field, and set the date with the current time:

```
SXMPUtils::ComposeStructFieldPath( kXMP_NS_SDK_EDIT, lastItemPath.c_str(),
    kXMP_NS_SDK_EDIT, "EditDate", &path );

XMP_DateTime dt;
SXMPUtils::CurrentDateTime( &dt );
meta.SetProperty_Date( kXMP_NS_SDK_EDIT, path.c_str(), dt );
```

4. Add the `EditComments` field to store an alt-text array for any localized versions of the document-update comments. Compose a path to the field and use `SetLocalizedText()` to set the value:

```
SXMPUtils::ComposeStructFieldPath( kXMP_NS_SDK_EDIT, lastItemPath.c_str(),
    kXMP_NS_SDK_EDIT, "EditComments", &path );

meta.SetLocalizedText( kXMP_NS_SDK_EDIT, path.c_str(),
    "en", "en-US", "Document created." );
```

5. Add the `EditTool` field:

```
meta.SetStructField( kXMP_NS_SDK_EDIT, lastItemPath.c_str(), kXMP_NS_SDK_EDIT,
    "EditTool", "FrameXML" );
```

Examining the new schema

To examine the schema we have built, we will dump the XMP object to a file.

1. Add code to serialize the XMP to RDF. See .

2. Add code to dump the XMP object to a file; to this, you add a callback function to dump the registered namespaces to a file. See .

3. Compile and run the application.

4. Review the output from the application. Notice that the namespaces for the custom schema have been registered. Here is an extract of the namespace dump with both the custom schema namespaces and the prefix used:

```
xmpT: => http://ns.adobe.com/xap/1.0/t/
xmpTPg: => http://ns.adobe.com/xap/1.0/t/pg/
xml: => http://www.w3.org/XML/1998/namespace
xmpDM: => http://ns.adobe.com/xmp/1.0/DynamicMedia/
xmpNote: => http://ns.adobe.com/xmp/note/
xmpidq: => http://ns.adobe.com/xmp/Identifier/qual/1.0/
xsdkEdit: => http://ns.adobe/xmp/sdk/Edit/
xsdkUser: => http://ns.adobe/xmp/sdk/User/
```

5. Examine the serialized RDF and dumped XMP and note how the properties are organized.