Adding Intelligence to Media

XMPFILES CUSTOM FILE-HANDLER PLUG-IN SDK

revision 2: June 2013

Adobe®

XMPFiles Custom File-handler Plug-in SDK

Adobe Systems Inc., 345 Park Avenue, San Jose, California 95110, USA.

Defining File-handler Plug-ins for XMPFiles

The XMPFiles library allows an application to handle XMP metadata supplied in a variety of file formats. The handlers for many file formats are built into the library. The API allows you to create an XMPFiles Plug-in that handles metadata for additional file formats, or replaces built-in format handlers with custom ones. XMPFiles automatically loads file-handler plug-ins from a location that you register when initializing the library, and treats them like the built-in format handlers.

SDK contents

The XMP Toolkit SDK includes the XMPFiles Plug-in SDK. This document assumes that you are familiar with the XMP Toolkit SDK; see the companion to this document, the *XMP Toolkit SDK Programmer's Guide*. You must build the XMPCore and XMPFiles libraries before you can build a plug-in that uses those libraries.

The `XMPFilesPlugins` folder includes the support code you need to build file-handler plug-ins for XMPFiles, as well as scripts that build a plug-in template project for your platform, which you can use to get started.

The download contains these folders:

`XMPFilesPlugins`	The root folder for the plug-in SDK.
`api`	Contains the C++ API. See "XMPFiles Plug-in API Reference" on page 17.
`PluginTemplate/build`	Contains the configuration and script files you need to create your own plug-in project in Windows, Mac OS, or Linux. See "Creating a plug-in template project" on page 6. The resulting project is a template that you can modify to create your own file-handler plug-ins for the XMPFiles library.
`CMakeLists.txt`	The `CMake` files used to create projects for each platform.
`resource/txt`	Configuration files for the project, including a unique identifier and a project manifest.
`GeneratePluginTemplate_win.bat`	Batch file for creating a plug-in template project for Visual C++ in Windows.
`GeneratePluginTemplate_mac.sh`	Script for creating an Xcode project in Mac OS.
`Makefile`	Make file for creating a GCC project in Linux.
`source`	Contains a template C++ file for a file handler. See "Implementing a file handler" on page 12
`PDFHandler`	Binary sources for the PDF handler.

New in this release

- "Plug-in versioning" on page 5

  This release of the XMPFiles Plug-in SDK supports backward and forward compatibility for plug-ins. It allows you to specify a version of host functionality that your plug-in uses, and loads the plug-in into the host only when the required functionality is available.

  This ensures, for example, that plug-ins created with a previous release continue to work with current and future versions of the libraries, and that plug-ins you create with current version can work in with previous XMP Toolkit releases (if they do not use any newly introduced functionality).

- "Creating a plug-in template project" on page 6

  This release does not include ready-to-use projects to use as templates; instead, it provides `cmakelist` files that build the plug-in template project for your platform using the `cmake` tool.

- API changes

  - New HandlerFlags value `kXMPFiles_NeedsPreloading`. If provided, the plug-in is loaded on initialization of the XMP Toolkit, regardless of whether the defined functionality is called for. Otherwise, it is loaded only when needed. This flag is not backward compatible; if you use it, your plug-in cannot be loaded in hosts that use a previous release of the XMP Toolkit.

  - New functions SetupPlugin() and RequestAPISuite() to support plug-in versioning.

  - New functions importToXMP() and exportFromXMP() to provide explicit mapping to and from non-XMP metadata formats.

  - New functions FillAssociatedResources() and IsMetadataWritable() to handle formats with multiple associated files.

  - Functions checkFormatStandard() and getXMPStandard() exposed to developers in the `StandardHandler` API suite (see "Plug-in versioning" on page 5).

    For a plug-in that provides a *replacement* handler, you can use these functions to check the format and retrieve the metadata using the original handler, if possible. A replacement handler is one for which the manifest Handler element's `Priority` attribute is `true`.

- Bug fixes

  This release fixes a number of multi-threading problems in the Plug-in SDK.

Plug-in versioning

The handling of XMP plug-ins has changed to make each version of the XMP Toolkit libraries compatible with a wider range of old and new plug-ins. You can now ensure that plug-ins created with a previous version of the libraries can be loaded and work in a current host, and that plug-ins you create with a new version can work with a previous host.

- Forward compatibility

  Plug-ins that have been created with a previous version of the XMP Toolkit libraries can be loaded with the current version. Similarly, plug-ins that you create with this version are guaranteed to be loadable in future versions.

  If a new host calls any functionality that is not present in the version with which a plug-in was created, the call simply returns an "unimplemented" error.

- Backward compatibility

  Plug-ins that you create with this version of the XMP Toolkit libraries can be loaded and can run in a previous version. Similarly, plug-ins created with future versions of the libraries will be loadable and can run in this version.

To ensure backward compatibility, this release offers a set of base functionality that is guaranteed not to change in future releases; this set of functions is always available to the plug-in through the `hostAPI` structure.

Starting with this release, new functions that are added to the SDK belong to individual, versioned *API suites*, defined in `hostAPI.h`. These suites can be modified in later versions of the SDK, and the modified suites will have higher version numbers. A plug-in can choose to include a specific version of an API suite by making a call to RequestAPISuite() from the SetupPlugin() initialization function. You can choose to prevent your plug-in from being loaded in a host that does not support the functionality it needs.

This mechanism ensures that your plug-in remains compatible with previous and future versions of the XMP Toolkit libraries, or is not loaded if it uses functionality that is not available in the current host.

Loading versioned suites

You must now implement the SetupPlugin() initialization function for your plug-in, whether or not you load any API suites. This function is called before the plug-in is loaded. It can simply return true if you do not need any additional suites. To request a specific versioned suite, your set-up function should call RequestAPISuite(), which returns a pointer to the suite if it is available, or NULL if not.

A plug-in that includes only the base functionality is guaranteed to be backward and forward compatible. If your plug-in depends on functionality from an API suite, you might not want to load your plug-in with an earlier version. If the suite you request is not available, your set-up function can return false to abort the load. (See this Setup example).

Always make sure that your plug-in uses the correct version of the XMP Toolkit libraries by linking statically to `XMPCore.lib`. If there is more than one handler for the same format available to the host, the one that calls for the latest version is loaded.

API suites

In this release, only one API suite is available:

Suite name	Version	Description
`StandardHandler`	2	You can use this API suite for a plug-in that defines a *replacement* handler. If the suite is available in the host, RequestAPISuite() returns a pointer to the structure containing function pointers. This suite defines the functions getXMPStandard() and checkFormatStandard(), which you can use to call the original handler to retrieve the XMP if possible.

Creating a plug-in template project

This release does not include ready-to-use projects to use as templates; instead, it provides a `cmake` tool that creates a plug-in template project for your platform. To create your project, execute the appropriate batch file, script, or makefile:

- In Windows
  - Run `GeneratePluginTemplate_win.bat` and choose from the option list.
  - The project is created in the `VC10\` folder.
- In Mac OS
  - Run `GeneratePluginTemplate_mac.sh` and choose from the option list.
  - The project is created in the `XCode/` folder.
- In Linux
  - Run `Makefile`.
  - The project is created in the `gcc/` folder, and the template plug-in is build in the `XMPFilesPlugins/public/` folder.

Project components

The project created by running the platform make file contains a unique identifier, or *module ID*. This is defined in the resource file `MODULE_IDENTIFIER.txt`, and retrieved by the global function `const char* GetModuleIdentifier()`.

You must modify the resource file to contain the unique identifier for your plug-in, and implement the retrieval function to return the same ID that is defined in the resource file.

In addition, the file `XMPPLUGINUIDS.txt` contains a *project manifest* that describes the plug-in content in XML format., using the following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<PluginResource Architecture="x86|x64">
   <Handler ...>
      <CheckFormat ... />
```

```
        <Extensions>
            <Extension Name="name">
            ...
        </Extensions>
        <FormatIDs>
            <FormatID Name="name"/>
        </FormatIDs>
        <HandlerFlags>
            <HandlerFlag Name="flag_constant"/>
            ...
        </HandlerFlags>
        <SerializeOptions>
            <SerializeOption Name="option_constant"/>
        </SerializeOptions>
    </Handler>
    ...
</PluginResource>
```

Project manifest elements

The project manifest contains these elements:

PluginResource

The root element.

```
<PluginResource Architecture="x86|x64">
```

ATTRIBUTES:

`Architecture`	▶ Required in Windows and Linux. Identifies the binary architecture of the contained handlers, one of "x86" (32-bit architecture) or "x64" (64-bit architecture). Handler plug-ins are loaded only if they have the same architecture as the XMPFiles library. For convenience, the PluginTemplate project contains both 32-bit and 64-bit versions of the sample manifest: `XMPPLUGINUIDS-32.txt` and `XMPPLUGINUIDS-64.txt`. You can select the one you need and rename it `XMPPLUGINUIDS.txt`. ▶ Optional in Mac OS, but must be omitted if the contained handlers are built as universal binaries. Handler plug-ins are loaded if they have the same architecture as the XMPFiles library, or if this attribute of the root element is omitted.

CONTAINS : A plug-in can contain one or more file-format handlers, each of which is described one Handler element in this collection.

Handler

Each element corresponds to a single file handler that implements a subclass of PluginBase.

```
<Handler  Name="fileHandlerName"
          Version="versionNumber"
          HandlerType="FolderHandler|OwningHandler|NormalHandler"
          Priority="true|false">
```

ATTRIBUTES:

`Name`	A unique identifying name for this handler.
`Version`	The version number for this handler, such as "1.0.0".
`HandlerType`	▶ `FolderHandler` for a folder-based handler. ▶ `OwningHandler` for a handler that takes responsibility for all file I/O for the handled format (that is, it does not use the built-in XMPFiles I/O functionality). ▶ `NormalHandler` for all other types of file handler.
`Priority`	True if this file handler replaces an existing built-in file handler for the same file format. Default is false. ▶ When true, if an existing handler for the format does not exist, the registration of this handler fails. ▶ A replacement handler can use the getXMPStandard() function to call upon the original handler's functionality, replacing that behavior only when needed.

CONTAINS : Required and optional elements:
Extensions,
FormatIDs

HandlerFlags,
SerializeOptions
CheckFormat

Extensions

Required if FormatIDs element is missing. A set of one or more elements , each of which associates this handler with a file name extension. These extensions are used to optimize the selection of file handlers. There is no requirement or guarantee that all files will use those extensions. When possible, the file contents are used to identify the file; see checkFileFormat() and checkFolderFormat().

```
<Extensions>
   <Extension Name="ext1"/>
   <Extension Name="ext2"/>
</Extensions>
```

FormatIDs

Required if Extensions element is missing. A set of one or more unique 4-byte file format constants. These are similar to the file format constants defined in `XMP_Const.h`, such as `kXMP_PDFFile`. Those values must be used when appropriate. For example, a plug-in for PDF must use `"PDF "`.

If an extension is specified in the `Extensions` element that is not defined in `XMP_Const.h`, you must include a corresponding `FormatID` element. For example, extension "xmp" must have a corresponding format ID, `"XMP "`.

```
<FormatIDs>
    <FormatID Name="4byte_id1"/>
    <FormatID Name="4byte_id2"/>
</FormatIDs>
```

HandlerFlags

Required. A set of one or more flags that identify the capabilities of this file handler. If you specify `kXMPFiles_CanInjectXMP` you must also specify `kXMPFiles_CanExpand`. For details of the handler capabilities, see XMPFiles documentation.

```
<HandlerFlags>
    <HandlerFlag Name="flag_constant"/>
    <HandlerFlag Name="flag_constant"/>
<HandlerFlags>
```

Flag constants are:

`kXMPFiles_CanInjectXMP`	`kXMPFiles_CanExpand`
`kXMPFiles_CanRewrite`	`kXMPFiles_PrefersInPlace`
`kXMPFiles_CanReconcile`	`kXMPFiles_AllowsOnlyXMP`
`kXMPFiles_ReturnsRawPacket`	`kXMPFiles_HandlerOwnsFile`
`kXMPFiles_AllowsSafeUpdate`	`kXMPFiles_NeedsReadOnlyPacket`
`kXMPFiles_UsesSidecarXMP`	`kXMPFiles_FolderBasedFormat`
`kXMPFiles_NeedsPreloading`	

SerializeOptions

Required, a set of option constants that specify how this file handler serializes the XMP packet. For details of the handler capabilities, see XMPFiles documentation.

```
<SerializeOptions>
   <SerializeOption Name="kXMP_UseCompactFormat"/>
</SerializeOptions>
```

Serialization option constants are:

`kXMP_OmitPacketWrapper`	`kXMP_ReadOnlyPacket`
`kXMP_UseCompactFormat`	`kXMP_UseCanonicalFormat`

kXMP_IncludeThumbnailPad	kXMP_ExactPacketLength
kXMP_OmitAllFormatting	kXMP_OmitXMPMetaElement
kXMP_EncodingMask	kXMP_EncodeUTF8
kXMP_EncodeUTF16Big	kXMP_EncodeUTF16Little
kXMP_EncodeUTF32Big	kXMP_EncodeUTF32Little

CheckFormat

Optional. If the file format can be identified by one or more byte sequences at a fixed location within the file, this element identifies those byte sequences. When XMPFiles checks a file format in order to determine which handler to use, it can use this information to identify the format before actually loading this plug-in. XMPFiles loads the plug-in only if the format matches all of these criteria.

```
<CheckFormat Offset="bytes" Length="bytes" ByteSeq="ASCII_or_hex"/>
```

ATTRIBUTES:

Offset	The offset of the beginning of the identifying sequence from the beginning of the file, in bytes.
Length	The length of the identifying sequence, in bytes.

ByteSeq	The specific identifying sequence. Declare the byte sequence either as an ASCII character string or as a hexadecimal number introduced by "0x":
	
	`<CheckFormat Offset="0" Length="4" ByteSeq="abcd"/>`
	`<CheckFormat Offset="10" Length="4" ByteSeq="0x8d25f621"/>`

Manifest example

A complete manifest file, as defined for the template plug-in, looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<PluginResource Architecture="x86">
<Handler
    Name="com.adobe.xmp.plugins.template"
    Version="1.00"
    HandlerType="NormalHandler"
    >
    <Extensions>
        <Extension Name="xmp" />
    </Extensions>
    <FormatIDs>
        <FormatID Name="TMP" />
    </FormatIDs>
    <HandlerFlags>
        <HandlerFlag Name="kXMPFiles_CanInjectXMP" />
        <HandlerFlag Name="kXMPFiles_CanExpand" />
        <HandlerFlag Name="kXMPFiles_CanRewrite" />
        <HandlerFlag Name="kXMPFiles_PrefersInPlace" />
        <HandlerFlag Name="kXMPFiles_CanReconcile" />
        <HandlerFlag Name="kXMPFiles_AllowsOnlyXMP" />
        <HandlerFlag Name="kXMPFiles_ReturnsRawPacket" />
```

```
        <HandlerFlag Name="kXMPFiles_AllowsSafeUpdate" />
    </HandlerFlags>
    <SerializeOptions>
        <SerializeOption Name="kXMP_UseCompactFormat" />
        <SerializeOption Name="kXMP_OmitPacketWrapper"/>
    </SerializeOptions>
</Handler>
</PluginResource>
```

Initializing XMPFiles to use your plug-in

The XMPFiles library does not define a default location for XMPFiles plug-ins. You must pass the location of any plug-ins you wish to use to the XMPFiles library during initialization. If you do not do so, XMPFiles does not load any plug-in file handlers.

Initialize()

Your program must call one of these initialization functions before using any other methods except `GetVersionInfo()`. The intialization functions are static; call them directly from the concrete class `SXMPFiles`.

```
bool Initialize ( const char* pluginFolder,
                        const char* plugins = NULL );
```

—or—

```
bool Initialize ( XMP_OptionBits options,
                        const char* pluginFolder,
                        const char* plugins = NULL );
```

PARAMETERS:

`options`	Optional. A logical OR of bit flags that control initialization. See XMPFiles documentation for details.
`pluginFolder`	The folder in which to find plug-in file handlers.
`plugins`	Optional. A comma-separated list of specific plug-in names. If supplied, XMPFiles loads only these plug-ins from the specified folder. Otherwise, all plug-ins found in the folder are loaded, as long as the architecture matches that of the XMPFiles library.

RETURN: True on success.

Implementing a file handler

To implement your own file handler, you can modify the provided template to customize the plug-in framework, then define the file-handling functionality for each file format you wish to support in a subclass of the SDK base class PluginBase.

Global initialization

You must implement these global functions:

- SetupPlugin()
- GetModuleIdentifier()
- RegisterFileHandlers():

SetupPlugin()

Called before your plug-in is loaded. Implement this function to request a specific version of any host API suite you need.

- If you do not need a versioned suite, or if the requested suite is available, this function should return true.
- If this requested suite is not available, this function should return false to abort the plug-in load operation.

Versioned suites contain additions to the base functionality that is available by default; see "Plug-in versioning" on page 5).

RequestAPISuite()

Call this helper function from your `SetupPlugin()` function to request additional API functions from the plug-in host, as provided in named, versioned API suites.

```
void* RequestAPISuite( char *apiName, XMP_Uns32 apiVersion )
```

PARAMETERS:

`apiName`	The name of the API suite.
`apiVersion`	The version number of the suite.

RETURN: A pointer to the requested suite structure if available; NULL if the specified version of the named suite is not found.

Setup example

```
bool SetupPlugin()
{
    // request the StandardHandler API suite, version 2
     StandardHandler_API_V2* standardHandler =
        reinterpret_cast<StandardHandler_API_V2*>(
               RequestAPISuite( "StandardHandler", 2 ) );
```

```
    //return true if suite available, false if not
     return (standardHandler != NULL);
}
```

GetModuleIdentifier()

Implement this function to return the Module ID as defined in the file `MODULE_IDENTIFIER.txt`. For example:

```
const char* GetModuleIdentifier()
{
    return "com.adobe.xmp.plugins.template";
}
```

RegisterFileHandlers()

Implement this function to register your file handler classes that are derived from PluginBase. The file handler's unique identifier must match the handler name that you have defined in the manifest. Each handler that a plug-in defines must be registered separately. For example:

```
//Register all the handlers provided by the plug-in.
void RegisterFileHandlers()
{
    PluginRegistry::registerHandler (
    new PluginCreator<Temp_MetaHandler> ( "com.adobe.xmp.plugins.template.handler1" )
);
}
```

Defining file handling

In order to interact with the XMP, you must link the XMPCore library to your plug-in and manipulate an object of the type `SXMPMeta`.

To implement a file handler, derive your handler class from PluginBase and implement the pure virtual methods to provide the basic functionality of reading and writing metadata in your file format:

- cacheFileData()

  When XMPFiles is opening a file of a type your handler supports, it calls the handler's implementation of this method. Your implementation should read the metadata from the file, coalesce it into XMP, and return the XMP as a UTF8 encoded string.

- updateFile()

  When XMPFiles needs to update the file, it calls the handler's implementation of this method. Your implementation should write all appropriate metadata back to the source file. XMPFiles calls this function only during the `SXMPFiles::CloseFile()` operation, and does not make any subsequent calls to access the metadata.

  If your handler is the owning handler, this function should respect the `doSafeUpdate` parameter and execute a safe update by writing to a temporary file and then swapping it with the original file.

Your class must provide additional static methods that are not defined in the base class:

- initialize()
  terminate()

  In your implementation of these functions, add any initialization and termination code that your file handler requires. There is no default initialization or termination behavior.

- checkFileFormat()
  checkFolderFormat()

  When XMPFiles is looking for a file handler to match a file format, it calls the static method `checkFileFormat()` for a file-based handler, or `checkFolderFormat()` for a folder-based handler. Your file handler must implement both methods. The method that does not match your handling type should simply return false.

Depending on your needs, you might also overwrite these virtual methods:

- importToXMP()
  exportFromXMP()

  If your file format contains metadata in non-XMP formats, you can implement import and export methods to reconcile the metadata. The `GetXMP()` and `CloseFile()` methods calls your handler's implementations of these functions to map the non-XMP values into and out of the XMP metadata.

- writeTempFile()

  If your handler supports crash-safe updating, but is NOT the owning handler, implement this method to rewrite the entire file content, including the XMP metadata, to an intermediate file, which XMPFiles swaps with the original file when the update is successful. XMPFiles calls your implementation of this method from the `CloseFile()` operation when doing a safe update.

  If your file format can have metadata in formats other than XMP, your handler might need to *export* it, mapping XMP values into the non-XMP when writing the metadata.

  If your handler is the owning handler, you must handle the safe-update option as part of your updateFile() implementation.

- getFileModDate()

  Implement this method to find the modification date of files in your handled format, if the default behavior is not sufficient.

Example class declaration

```
class MyHandler : public PluginBase
{
public:
              MyHandler( const std::string& filePath );
             ~MyHandler();
   /**
    * Load XMP metadata from the passed source file
    *
    * @param file    I/O interface
    * @param xmpStr  [out] Return the XMP metadata as UTF8 encoded string
    */
   virtual void cacheFileData( const IOAdapter& file, std::string& xmpStr );
```

```
 /** Reconcile the XMP and non-XMP metadata values here. This is called by GetXMP().
  *
  * @param xmpStr A pointer to the string containing the raw XMP Packet
  */
 virtual void importToXMP( XMP_StringPtr* str);

/**
 * Update metadata in the file using the passed I/O interface
 *
 * @param file           I/O interface with which to access the file
 * @param doSafeUpdate   Do a safe update (store in a temp file first)
 * @param xmpStr         The XMP metadata as UTF8 encoded string
 */
virtual void updateFile( const IOAdapter& file, bool doSafeUpdate,
                         const std::string& xmpStr );

 /**
  * Reconcile the XMP with non-XMP metadata values and update any non-XMP metatadata
  * in the file here. This is called by CloseFile().
  *
  *@param xmpStr A pointer to the string containing the XMP Packet to be written
 */
 virtual void exportFromXMP( XMP_StringPtr xmpStr );

/**
 * Do a safe update of the original file by writing a temp file that XMPFiles
 * can swap for the original
 *
 * @param srcFile     I/O interface to source file
 * @param tmpFile     I/O interface to temp file
 * @param xmpStr      The XMP metadata as UTF8 encoded string
 */
virtual void writeTempFile( const IOAdapter& srcFile, const IOAdapter& tmpFile,
                            const std::string& xmpStr );

 /** Retrieve the modification date/time of the metadata file
  *
  * @param modDate [out] A buffer in which to return the modification date.
  * @return        True if a modification date could be determined
  *
  */
virtual bool getFileModDate ( XMP_DateTime * modDate );

/**
 * Initialize the file handler
 * This method is called once during loading the plugin. Any required initialization
 * related to the file handler can be added here.
 *
 * @return  true on success
 */
static bool initialize();

/**
 * Terminate the file handler
 * This method is called once during unloading the plugin. Any required termination
 * related to the file handler can be added here.
 *
 * @return  true on success
 */
static bool terminate();
```

```
    /**
     * Check the file format.
     * Called while finding a handler for a specific file format.
     * Called for file-based handlers only. Not called if the manifest provides one or
     * more <CheckFormat> elements
     *
     * @param filePath  Path to a source file
     * @param file      I/O interface to source
     * @return  true if the file handler support the file format
     */
    static bool checkFileFormat( const std::string& filePath, const IOAdapter& file );

    /**
     * Check the file format.
     * Called while finding a handler for a specific file format.
     * Called for folder-based handlers only.
     *
     * @param rootPath  Root of the folder format
     * @param gpName Grandparent of the leaf file
     * @param parentName Parent of the leaf file
     * @param leafName Name of the leaf file
     * @return  true if the file handler support the file format
     */
    static inline bool checkFolderFormat( const std::string& rootPath,
                                          const std::string& gpName,
                                          const std::string& parentName,
                                          const std::string& leafName );
};
```

XMPFiles Plug-in API Reference

The API defines these base classes that you use to create file handler plug-ins for XMPFiles:

- PluginBase: The base class for file-handler plug-ins.
- IOAdapter: The interface for reading from and writing to data sources.

PluginBase

All new file handlers must derive from this base class. Some of the methods must or can be specialized to provide your file-handling functionality, and some provide supporting functionality.

In order to manipulate an XMP packet in your implementations of these methods, your plug-in must link to the XMPCore library that defines the `SXMPMeta` type.

The base class defines these methods (presented here alphabetically):

Method	Description	Implement in plug-in
cacheFileData()	For a file-based handler, implement this method to read the XMP metadata from the file, import any non-XMP values into the XMP, and return the XMP as a UTF8 encoded string.	Required
checkAbort()	Allows the plug-in to abort the current operation.	No
checkFileFormat()	For a file-based handler, implement this method to look into the contents of the file in order to identify whether it is in a format that this handler can process. For a folder-based handler, implement this method to return false.	Required
checkFolderFormat()	For a folder-based handler, implement this method to determine whether the folder should be handled. For a file-based handler, implement this method to return false.	Required
checkFormatStandard()	A replacement handler can call to check the format using the original handler's check-format method.	No
exportFromXMP()	Reconciles XMP and non-XMP metadata values; called by `CloseFile()`.	Optional
FillAssociatedResources()	Retrieves a list of resources associated with the open file.	Optional
getFileModDate()	If the default behavior does not work for your file format, implement this method to return a meaningful value for the file modification date.	Optional

Method	Description	Implement in plug-in
getFormat()	Retrieves the file format identifier for which the current instance was created.	No
getHandlerFlags()	Retrieves the flags that identify the capabilities of this handler.	No
getOpenFlags()	Retrieves the options that describe the desired access.	No
getPath()	Retrieves the path to the input file or folder for which this handler was called.	No
getXMPStandard()	A replacement handler can call to retrieve XMP metadata using the original file handler.	No
importToXMP()	Reconciles XMP and non-XMP metadata values; called by `GetXMP()`.	Optional
initialize()	Implement this method to perform any initialization that your file handler requires.	Required
IsMetadataWritable()	Reports whether metadata can be updated in the open file.	Optional
terminate()	Implement this method to perform any termination cleanup that your file handler requires.	Required
updateFile()	Implement this method to export metadata into any non-XMP metadata formats, and write the metadata back to the source file.	Required
writeTempFile()	If your handler supports crash-safe updating and is not the owning handler, implement this method to write the file content, including the XMP metadata, to an intermediate file.	Optional

cacheFileData()

Implement this method to read the XMP metadata from the file and return it as a UTF8 encoded string. When XMPFiles is about to open a file of a type your handler supports, it calls the handler's implementation of this method.

```
void cacheFileData( const IOAdapter& file,
                    std::string& xmpStr );
```

PARAMETERS:

`file`	The I/O interface that provides access to the source file.
`xmpStr`	A string in which to return the XMP read from the file, in UTF8 encoding.

checkAbort()

Allows XMPFiles to abort the current operation. XMPFiles calls this to check whether the current operation is in a state that can be aborted.

```
bool checkAbort( bool doAbort = false );
```

PARAMETERS:

`doAbort`	▶ When true, this method throws the "User abort" exception if the operation can and should be aborted, instead of returning a Boolean value. ▶ When false or not supplied, XMPFiles throws the exception if this method returns true.

RETURN: True if the current operation can and should be aborted; false otherwise.

checkFileFormat()

For a file-based handler, implement this method to look into the contents of the file in order to identify whether it is in a format that this handler can process. Typically, a handler reads a file to look for a specific byte sequence that uniquely identifies the format. Use the I/O interface file to read bytes that would uniquely identify the format, and return true if a match is found.

```
bool MyHandler::checkFileFormat(  const std::string& filePath,
                                  const IOAdapter&   file )
```

PARAMETERS:

`filePath`	The path to the data source file.
`file`	The I/O interface that provides access to the source file.

RETURN: True if this is a file-based handler and the file matches the handled format; false otherwise.

NOTE: The version of XMPFiles included in the SDK differs from an earlier version that is built into the initial release of CS6 applications. In the earlier version, handlers of type "NormalHandler" never call `checkFileFormat()` unless the CheckFormat manifest entry is present; this can result in unexpected problems, depending on the file extension of the opened file and format ID of the handler. It is therefore recommended that you always include the `CheckFormat` entry in the manifest, if possible.

Later versions of CS6 applications (CS 6.0.1) incorporate the later version of XMPFiles; check with technical support for the latest version information.

checkFolderFormat()

For a folder-based handler, implement this method to determine whether the folder should be handled. For a file-based handler, always return false.

For information about folder-based handling, see XMPFiles documentation.

```
bool MyHandler::checkFolderFormat(   const std::string& rootPath,
                                     const std::string& gpName,
                                     const std::string& parentName,
                                     const std::string& leafName )
```

PARAMETERS:

`rootPath`	The path to the folder, as defined for the format.
`gpName`	The grandparent of the leaf file.
`parentName`	The parent of the leaf file.
`leafName`	The file name of the leaf file to be handled.

RETURN: True if this is a folder-based handler and the folder matches the handled format; false otherwise.

EXAMPLE: For P2 format, when checking the format of the file `.../MyMovie/CONTENTS/CLIP/0001AB.XML`, the parameters are:

```
root: .../MyMovie
gpName: CONTENTS
parentName: CLIP
LeafName: 0001AB.XML
```

checkFormatStandard()

Calls the check-format method of the standard file handler to check the file format of the data source. This should be used only by a replacement handler; if called by any other kind of handler, fails with an exception. A replacement handler is one for which the manifest Handler element's `Priority` attribute is `true`.

This function is part of the `StandardHandler` API suite; see "Plug-in versioning" on page 5.

Called by getXMPStandard().

```
bool checkFormatStandard( const std::string* path = NULL )
```

PARAMETERS:

`path`	Pointer to the path string of the data source file to check, or NULL to check the fie passed during initialization.

RETURN: True on success.

exportFromXMP()

XMPFiles calls this method from `CloseFile()` when writing metadata back to the open file. Implement it to reconcile and update any non-XMP metadata.

```
void MyHandler::exportFromXMP( XMP_StringPtr xmpStr )
```

PARAMETERS:

`xmpStr`	The string containing the XMP Packet to be written.

RETURN: Nothing.

FillAssociatedResources()

Retrieves all resource files, XMP or non-XMP, associated with the open file. The default implementation only considers the case of a typical single file with embedded metadata, and returns that one file. For all other cases, you must implement this method to return the correct set of files.

- For a file format that might have a sidecar XMP file, your implementation should return the path of the file itself if there is no sidecar file, or two paths if a sidecar XMP file exists.
- For a folder-based handler, your implementation should return the paths of all associated files, including the files and folders necessary to identify the format. All returned paths must exist. The root path must be the first entry in the returned vector.

```
void MyHandler::FillAssociatedResources ( std::vector<std::string>* resourceList );
```

PARAMETERS:

`resourceList`	A pointer to a vector that the method populates with the list of associated resources.

RETURN: Nothing.

getFileModDate()

XMPFiles calls this method when it needs to report the modification date of the data source file or folder. The default implementation returns the modification date of the file with which the instance of PluginBase was initialized. If this is not the correct behavior, your implementation should return a meaningful value for your file format. The method should return true if a modification date can be determined, false otherwise.

The default implementation always returns false if any of these flags are set:

```
kXMPFiles_HandlerOwnsFile
kXMPFiles_UsesSidecarXMP
kXMPFiles_FolderBasedFormat
```

If your handler sets any of these flags but can retrieve a modification date, you must supply your own implementation of this method in order to do so.

```
bool  MyHandler::getFileModDate( XMP_DateTime* modDate );
```

PARAMETERS:

`modDate`	A date-time structure in which to return the modification date.

RETURN: True on success. False if the date retrieval fails for any reason.

getFormat()

Retrieves the file format identifier for which the current instance was created.

```
XMP_FileFormat getFormat() const;
```

RETURN: A file-format constant, as defined in XMPFiles. See XMPFiles documentation for details.

getHandlerFlags()

Retrieves the flags that identify the capabilities of this handler. See XMPFiles documentation for details.

```
XMP_OptionBits getHandlerFlags() const;
```

RETURN: A logical OR of the bit-flag constants. Flag constants are:

kXMPFiles_CanInjectXMP	kXMPFiles_CanExpand
kXMPFiles_CanRewrite	kXMPFiles_PrefersInPlace
kXMPFiles_CanReconcile	kXMPFiles_AllowsOnlyXMP
kXMPFiles_ReturnsRawPacket	kXMPFiles_HandlerOwnsFile
kXMPFiles_AllowsSafeUpdate	kXMPFiles_NeedsReadOnlyPacket
kXMPFiles_UsesSidecarXMP	kXMPFiles_FolderBasedFormat
kXMPFiles_NeedsPreloading	

getOpenFlags()

Retrieves the options that describe the desired access from the `SXMPFiles::OpenFile()` operation. See XMPFiles documentation for details.

```
XMP_OptionBits getOpenFlags() const;
```

RETURN: A logical OR of the bit-flag constants. Option constants are:

```
kXMPFiles_OpenForRead
kXMPFiles_OpenForUpdate
kXMPFiles_OpenOnlyXMP
kXMPFiles_OpenUseSmartHandler
kXMPFiles_OpenUsePacketScanning
kXMPFiles_OpenLimitedScanning
```

getPath()

Retrieves the path to the input file or folder for which this handler was called.

```
const std::string& getPath() const;
```

RETURN: The absolute path string, or an empty string if the data source is neither a file nor folder.

getXMPStandard()

Retrieves XMP metadata using the default file handler. This should be used only by a replacement handler; if called by any other kind of handler, it fails with an exception. A replacement handler is one for which the manifest Handler element's `Priority` attribute is `true`.

This function is part of the `StandardHandler` API suite; see "Plug-in versioning" on page 5.

This call uses checkFormatStandard() to check the file format, and retrieves the metadata only if that call returns true.

```
bool getXMPStandard( std::string& xmpStr,
                     const std::string* path = NULL,
                     bool* containsXMP = NULL);
```

PARAMETERS:

`xmpStr`	A string reference that is populated with the XMP Packet read by the standard handler if the call succeeds.
`path`	Pointer to the path of the data source file to check, or NULL to check the fie passed during initialization; see getPath().
`containsXMP`	Returns true if the standard handler detected XMP metadata.

RETURN: True on success.

importToXMP()

XMPFiles calls this method from `GetXMP()` when reading metadata from the open file. Implement it to reconcile the raw XMP Packet with any non-XMP metadata in the file.

```
void MyHandler::importToXMP( XMP_StringPtr* xmpStr )
```

PARAMETERS:

`xmpStr`	A pointer to the string pointer containing the raw XMP Packet.

RETURN: Nothing.

initialize()

XMPFiles calls the `initialize()` function of each file handler once when it loads the plug-in.

You must implement this as a static method. Your implementation should add any initialization code that your file handler requires.

```
bool MyHandler::initialize()
```

RETURN: True on success. When this method returns false, XMPFiles cannot access the plug-in.

IsMetadataWritable()

Reports whether metadata can be updated or written to the format. The default implementation only handles the case of a single file with embedded metadata.

For folder-based video formats, your implementation should return true only if metadata can be written in all of the related metadata files.

```
bool MyHandler::IsMetadataWritable()
```

RETURN: True if metadata is writeable, false otherwis

terminate()

XMPFiles calls the `terminate()` function of each handler once when it unloads the plug-in (that is, when the XMPFiles library itself is terminated).

You must implement this as a static method. Your implementation should add any termination code that your file handler requires.

```
void MyHandler::terminate()
```

updateFile()

When XMPFiles is about to close the file, it calls this method. Your implementation should write XMP metadata back to the source file.

```
virtual void MyHandler::updateFile(  const IOAdapter& file,
                                     bool doSafeUpdate,
                                     const std::string& xmpStr );
```

PARAMETERS:

`file`	The I/O interface that provides access to the source file.
`doSafeUpdate`	If this handler is the owning handler and this is true, your implementation must perform a safe update by writing the metadata to a temporary file, then swapping that for the original source file. If your handler implements crash-safe updating in this method, indicate this by setting the HandlerFlags `kXMPFiles_AllowsSafeUpdate` and `kXMPFiles_HandlerOwnsFile` in the manifest. If this is not the owning handler but supports safe-update, XMPFiles calls your handler's writeTempFile() method when safe-update is required.
`xmpStr`	The string containing the XMP metadata to be written.

writeTempFile()

If your handler supports crash-safe updating, can update the whole file (as indicated by the `kXMPFiles_CanRewrite` flag) and is not the owning handler, XMPFiles calls this method to write the entire file content, including the XMP metadata, to an intermediate file when it is about to close the data source file using the safe-save option. See `SXMPFiles:CloseFile()` and the `kXMPFiles_UpdateSafely` option.

If your file format can have metadata in formats other than XMP (such as EXIF), your handler is responsible for *exporting* it; that is, mapping XMP values into the other formats when writing the metadata.

If your handler implements crash-safe updating, set the Handler flags `kXMPFiles_AllowsSafeUpdate` and `kXMPFiles_CanRewrite` in the manifest. If your handler does not support safe-update, XMPFiles attempts to perform its default implementation, which might not be the best solution for your file format.

```
void MyHandler::writeTempFile( const IOAdapter& srcFile,
                               const IOAdapter& tmpFile,
                               const std::string& xmpStr );
```

PARAMETERS:

`srcFile`	The I/O interface that provides access to the source file.
`tmpFile`	The I/O interface that provides access to the temporary file.
`xmpStr`	The string containing the XMP metadata to be written.

NOTE: The version of XMPFiles included in the SDK differs from an earlier version that is built into the initial release of CS6 applications. This method can only be used with the later version included in the SDK; that is, in a plug-in developed for a third-party application that incorporates the XMPFiles library provided with the SDK.

Later versions of CS6 applications (CS 6.0.1) incorporate the later version of XMPFiles; check with technical support for the latest version information. If your plug-in runs in a CS6 application or extension without the patch, this method does not work as expected. XMPFiles does not pass the XMP Packet to the plug-in handler.

IOAdapter

This interface provides data reading and writing functionality. An object of this type allows you to perform the operations you need for the data source your plug-in handles, which can be a file or any other possible source defined by the host system and XMPFiles client application.

The interface defines these methods (presented here alphabetically):

AbsorbTemp()	Replaces the original content of the current data source with content of a temporary file at the end of a successful safe-save operation.
DeleteTemp()	Deletes the temporary file used in a failed safe-save operation, leaving the original data source unchanged.
DeriveTemp()	Creates a temporary file for a safe-save operation.
Length()	Reports the length of the current data source.
Read()	Reads data from the current data source into a buffer.
Seek()	Set the I/O position in the current data source.
Truncate()	Truncates the current data source to a given length.
Write()	Writes data from a buffer to the current data source.

AbsorbTemp()

Replaces the original content of a data source with content of a temporary file at the end of a successful safe-save operation. Closes and deletes the temporary file after the replacement operation is completed; see DeriveTemp().

```
void AbsorbTemp();
```

ON ERROR: Throws the exception `XMPError` if the temporary file cannot be absorbed.

DeleteTemp()

Deletes the temporary file used in a failed safe-save operation, leaving the original data source unchanged. Call this if AbsorbTemp() throws an error; see DeriveTemp().

```
void DeleteTemp();
```

ON ERROR: If no temporary file exists, does nothing.

DeriveTemp()

Creates and returns a temporary file for a safe-save operation. This is normally associated in some way with the original data source; for example in the same directory and with a related name.

This can return an existing temporary `XMP_IO` object or create a new one. The temporary file must be opened for read-write access for use in a safe-save operation, which uses portions of the original file and adds new data to the temporary file, then swaps it for the original file when the update has succeeded. This method throws an exception if the owning object is read-only, or if it cannot create the temporary file.

The temporary file is normally closed and deleted, and the temporary `XMP_IO` object deleted, by a call to AbsorbTemp() or DeleteTemp(). Use the derived `XMP_IO` object's destructor if necessary.

```
XMP_IORef DeriveTemp();
```

RETURN: A pointer to the temporary `XMP_IO` object for the new file.

ON ERROR: If the owning object is open for read-only access, or if the function cannot create a new object, throws an `XMPError` exception.

Length()

Reports the length of the current data source at the current I/O position, in bytes. The I/O position remains unchanged.

```
XMP_Int64 Length();
```

RETURN: The length of the file in bytes.

Read()

Reads data from the current data source into a buffer, returning the actual number of bytes read.

```
XMP_Uns32 Read(  void* buffer,
                 XMP_Uns32 count,
                 bool readAll );
```

PARAMETERS:

`buffer`	A pointer to the buffer.
`count`	The length of the buffer in bytes.
`readAll`	True if reading less than the requested amount is considered failure.

RETURN: The number of bytes read.

ON ERROR: If *`readAll`* is true and not enough data is available, throws an `XMPError` exception; in this case, the buffer content and I/O position are undefined.

Seek()

Sets the I/O position in the current data source, returning the new absolute offset in bytes. A seek beyond EOF is allowed when writing, and extends the file. This is equivalent to seeking to EOF then writing the needed amount of undefined data.

```
void Seek( XMP_Int64& offset,
           SeekMode mode );
```

PARAMETERS:

`offset`	The offset relative to the mode. Can be positive or negative.
`mode`	The origin of the seek operation. See XMPFiles documentation for details.

RETURN: The new absolute offset in bytes.

ON ERROR: If the file is read-only, and the seek results in a position beyond EOF, throws an `XMPError` exception..

Write()

Writes data from a buffer to the current data source at the current I/O position, overwriting existing data and extending the file as necessary.

```
void Write( void* buffer, XMP_Uns32 count );
```

PARAMETERS:

`buffer`	A pointer to the buffer.
`count`	The length of the buffer in bytes.

ON ERROR: If all data cannot be written, throws an `XMPError` exception..

Truncate()

Truncates the current data source to a given length. The I/O position after truncation remains unchanged if still valid; otherwise sets it to the new EOF.

```
void Truncate( XMP_Int64 length );
```

PARAMETERS:

`length`	The new length for the file, which must be less than or equal to the original length.

ON ERROR: If the new length is longer than the file's current length, throws an `XMPError` exception.