Working with Metadata in ASF Files Using the Windows Media Format SDK

By Jay Loomis[1]

[1] This document is a compilation and revision of information originally written for the Windows Media Format SDK documentation version 9.5. I am almost entirely responsible for that documentation, and have created this document based upon it specifically as a writing sample.

Contents

Introduction

This document describes how to use the objects of the Windows Media Format SDK to inspect and manipulate metadata associated with Advanced Systems Format (ASF) files (typically .wma audio or .wmv video files). Though all relevant use of Windows Media Format SDK application programming interfaces (APIs) is explained in this document within the context of metadata operations, the broader context of ASF file use is not covered. You will find the Windows Media Format SDK documentation essential to understanding what is written here.

Metadata in ASF Files

Metadata is, by definition, data about data. In the case of an audio or video file, metadata can describe either properties of the encoded media in the file, such as the number of frames or samples per second, or characteristics of the original media content, such as actors or musicians performing in the work. This document deals exclusively with the latter type of metadata, often called *content-descriptive* metadata.

Individual items of metadata are called metadata attributes. They are stored in the header[2] of an ASF file as name/value pairs. With a very few exceptions, the ASF specification does not define particular attributes. While this means that content creators are able to use whatever attributes they like, it would be of little use to consumers if every file creator included a different set of attributes. As part of the Format SDK, Microsoft defines a large number of standard attributes, and provides global constants for the name of each so that you don't need to worry about misspelling an attribute (or, rather, so that if you do misspell an attribute, the error will be caught at compile time).

There are three categories of content-descriptive metadata: core attributes, extended attributes, and custom attributes. Let's look at the particulars of each category.

Core attributes

Core attributes are those that are defined in the ASF specification. These most basic attributes that describe content in and audio or video file. The following table shows the five core attributes:

Attribute	SDK Constant	Description
title	g_wszWMTitle	The title of the content
author	g_wszWMAuthor	The name of the primary author of the content
copyright	g_wszWMCopyright	A copyright message about the content
description	g_wszWMDescription	A description of the content
rating	g_wszWMRating	User rating of the content

There are no reliable standards regarding the use of the core attributes, so you may encounter them with inconsistent formatting in files that you did not encode. For example, some applications may represent the user rating as a value from one to five, while others may use only three (as with many

[2] The ASF header object is the first object in an ASF file. The other required top-level object is the data object that contains the media stream data. In practice, ASF files usually have an index object as well, that makes seeking within streams possible.

modern music services that enable you to give a song a “thumbs up” if you like it, a “thumbs down” if you don’t, and must capture a third value for songs you have not rated). If you are creating an application to play files that you create yourself (or files that you populate with your own metadata), you may want to create custom attributes that duplicate the functionality of the core attributes. Doing so enables you to ensure that the attributes you display are formatted to your own specification. For example, if I were writing a music player, I might decide that I want to capture the ratings for each song using some predictive algorithm so that, in addition to the thumbs up, thumbs down, and not rated states, I’ll add “likely to like” and “likely to dislike” values. When naming such a custom attribute, it can be helpful to use a consistent prefix to avoid confusion with other similar custom attributes—so I could prepend my initials to my custom rating name (e.g. JLRating).

NOTE You should always populate the core metadata attributes, even if doing so is redundant with your custom attributes. It is a better user experience to find metadata for files when used in another application than to have no attributes display.

Extended Attributes

The extended attributes are those that are defined as constants in the Windows Media Format SDK. These are attributes that files need in order to be compatible with Windows Media Player and to be more or less in compliance with the ID3 metadata definition (in common use for MP3 files). Many extended attributes are defined for your use. You can find a list of all the defined attributes in the Windows Media Format SDK documentation.

Note that, while we’ve drawn a contrast between the core and extended attributes for the purpose of this discussion, the defined attributes of both types are treated the same when using the objects of the Format SDK to manipulate them. The primary difference between the core and extended attributes is that, while you should always set values for the core attributes, you should not feel any particular obligation to attempt to “complete” a file’s metadata by assigning values for all of the potentially applicable extended attributes. Rather, you should use the extended attributes when they apply to your content and are appropriate to the needs of your application.

Custom Attributes

You may define and use any custom attributes that you need to support your application or service. There is no practical difference between any custom attributes you create and the core and extended attributes that are defined in the Format SDK: in both cases, attributes are name-value pairs stored in the header of the ASF file. Bear in mind that you should use the provided string constants in the Format SDK to refer to the names of core and extended attributes, which will give you compile-time warnings if you accidentally type the name incorrectly. You should create constants for your custom attributes when assigning them programmatically, to gain the same benefit. The other consideration with custom attributes is that other applications that work with your files will most likely do nothing with them. If you are supporting an application that uses exclusive files that will never be loaded by other programs, you need not be concerned with the essentially hidden nature of your custom attributes.

Attribute Data Types

ASF files, including their headers, are binary data, not text data. As such, each value is stored using the number of bytes that corresponds to that data type in C++.

Attribute values can be formatted as one of a handful of data types, and take up a number of bytes in the file header according to their type. In addition to several sizes of integer values, Unicode strings of variable size, and other common data types, attributes can be arbitrary binary arrays. In practice, such arbitrary data should always correspond to a data structure. The variable nature of attribute values requires some extra steps when retrieving them from files, as we'll see when we discuss using the objects of the Windows Media Format SDK.

Attributes with Multiple Values

ASF files can have multiple attribute values for the same name. Sometimes this can be a systemic decision, such as creating a file with metadata in multiple languages (for example, including both regional English and regional French in files for consumption in Canada). Other times, specific attributes may need more than one value to be complete.

In versions of the Windows Media Format SDK prior to support for multiple values, content encoders and metadata services tended to work around the issue in an ad hoc manner. The biggest problem was with the artist attribute: what name do you put in this attribute for a song? People would put many names in a single string, sometimes delimiting them with commas, but other times using semicolons or other characters. The lack of consistency meant that, at worst, player applications couldn't parse the artist and displayed garbled results. Now, you can assign multiple values to the same name and reliably retrieve them as individual values—no formatting guesswork required.

Applications of ASF Metadata

Most of the metadata attributes that are predefined for use with the objects of the Windows Media Format SDK are informational. You use them to store facts about the content that is encoded into the file, like the name of the artist who created it or the date of publication. The primary purpose of this kind of information is to be displayed for an end user in a media playing application. However, the structure of an ASF file is flexible enough to support other uses as you see fit. You can create custom metadata attributes to include in files and then add code to a custom application that responds to them.

As with most aspects of the highly flexible ASF specification, the typical use case for metadata is very simple. It is up to you to determine whether the flexibility of the format can be used to enhance your applications. Be aware that other software may place custom metadata in ASF files when opening them, so you should never assume what attributes will or will not be present unless you only deal with files you have encoded yourself.

An example of something that you might want to do with custom metadata is synchronizing advertisements to specific points in a file. Such functionality could be useful if your advertising strategy for a music player was to associate a set of advertisements to each song. You could create one or more custom metadata attributes to hold the playback time and associated advertisement URL. Then you would extract the advertisement information from each file when you load it into your player and write

code that checks the time stamp of the playing file in order to present the advertisements at the desired frequency and in the right sequence. This example is intended to give you an idea of the potential power of ASF metadata, but in practical terms the functionality described would probably be better implemented using script commands, or by Web streams. You can read more about both of those features in the Format SDK documentation.

Metadata in Format SDK Objects

You manage content-descriptive metadata with the **IWMHeaderInfo3** interface, which is implemented by the reader object, the synchronous reader object, the writer object, and the metadata editor object. We will only use the metadata editor object for the examples in this document. Refer to the Format SDK documentation if you want to work with metadata from an interface of one of the other supported objects, as the availability of some attributes varies from one object to another.

In order to use the methods of **IWMHeaderInfo3**, you need to do the following:

1. Initialize COM.
2. Create an instance of the metadata editor object by calling the **WMCreateEditor** helper function.
3. Get a pointer to **IWMHeaderInfo3** from the created object by calling **QueryInterface** and passing it the IID_IWMHeaderInfo3 GUID constant.

The following code demonstrates this process[3]:

```
HRESULT hr = S_OK;
IWMMetadataEditor *pEditor = NULL;
IWMHeaderInfo3    *pHeaderInfo = NULL;

hr = CoInitialize(NULL);
// Be sure to perform appropriate error checking. You can use the
//  FAILED macro for the simplest checks:
if(FAILED(hr)){
   // Do something useful.
}
hr = WMCreateEditor(&pEditor);
hr = pEditor->QueryInterface(IID_IWMHeaderInfo3, &pHeaderInfo);
```

[3] In this code snippet, and in most other code in this document, error checking has been elided for the sake of concision and clarity. At a minimum, your code should check each HRESULT value returned by COM methods and handle them as dictated by the needs of your application. Always remember to release COM interfaces when you are finished with them and to de-allocate any dynamic memory. The Windows Media Format SDK documentation describes some common methods for error checking if you need more information.

Attributes with Multiple Values and Attribute Indices

The methods of the **IWMHeaderInfo3** interface present a programming model that replaces an older one defined in **IWMHeaderInfo**, which enabled you to search for an attribute by name or by index. The old way of doing things is obsolete because many attributes are now allowed to have multiple instances in a file, and you can never be sure of getting a particular instance of an attribute if you search by name. Support for multiple values was added because some values might require more than one entry. In earlier versions of the SDK, content encoders would get around the limitations of the attributes by putting multiple names into a single attribute, delimited by commas or semicolons. As you can imagine, packing multiple names into one attribute without any enforced standard of delineation led to difficulties when files were played in different player applications. The new model requires slightly more complicated code, but enables more consistent handling of attributes without trying to enforce arbitrary standards.

The new way of getting attributes is to use an index of all those attributes that match a specified set of search criteria. The metadata editor object dynamically creates an index of matching attributes when you ask for one, so the indices are never stored in the file itself. See Reading Metadata in this document for specific instructions to get attributes from a file.

File Management

The methods of the **IWMMetadataEditor** interface, which is the interface pointer you got from the **WMCreateEditor** helper function, deal with file operations. The following topics describe how to use those methods to load and close a file.

Open a File

You use the **Open** method to load a file into the metadata editor. It takes only one argument, which is the name of the file to be opened:

```
WCHAR *pwszFile = NULL;
// Write code here to allocate the file name string and get the
//  name, as appropriate to your application.
hr = pEditor->Open(pwszFile);
```

Please note that the **Open** method does not include a string length argument, so it is extremely important that you only pass a NULL-terminated string.

This basic file opening method should work for most simple applications. If you need to use file sharing, you must use the **OpenEx** method, which is implemented in the **IWMMetadataEditor2** interface. You can get a pointer to the **IWMMetadataEditor2** interface of your metadata editor object by calling **QueryInterface**.

Save Changes to a File

When you open a file into the metadata editor object, you begin a session with that file. While you inspect and change attributes in the file, the metadata editor tracks changes internally. The changes are

only made to the file itself when you tell the metadata editor object to do so. The process of saving your changes to the file is called a flush, as in flushing the changes from memory to the file. The **Flush** method of the **IWMMetadataEditor** interface does this for you.

```
// We'll presume you've done something interesting with the
//  metadata here.
hr = pEditor->Flush();
// Now the editor is ready to load the next file, if needed.
```

Flush also closes the file. If you are making an application that needs to keep a file open for a long time and you want to save your changes incrementally, you will need to periodically flush the changes and then reopen the file.

Close a File Without Saving

Sometimes you have no changes to save. If that is the case, you can close the file with the **Close** method:

```
hr = pEditor->Close();
```

Metadata Operations

Once you have created a metadata editor object, opened a file into it, and retrieved a pointer to the **IWMHeaderInfo3** interface, you are ready to start working with metadata. The following topics describe how to use the methods of **IWMHeaderInfo3** to manipulate attributes.

Understanding Stream Numbers

Metadata attributes can be assigned to individual streams in the file or to the file in general. When you query for attributes you must tell the metadata editor object what stream you want to look for. You can use a value of zero to get only attributes that are set at the file level. You can also use a value of 0xFFFF to get all attributes that match your query regardless of their stream affiliation. Retrieving stream numbers in ASF files is outside the scope of this document, so we will focus on file-level and universal queries.

Working with Attribute Data Types

Every attribute in an ASF file is assigned a data type using a value from the enumeration, WMT_ATTR_DATATYPE. You can find a full description of the enumeration in the Format SDK documentation, but the following table summarizes the possible values:

Enumeration	Data type information
WMT_TYPE_DWORD	An integer of type DWORD
WMT_TYPE_STRING	A NULL-terminated Unicode string
WMT_TYPE_BINARY	A byte array corresponding to a structure
WMT_TYPE_BOOL	A Boolean value of type BOOL
WMT_TYPE_QWORD	An integer of type QWORD
WMT_TYPE_WORD	An integer of type WORD
WMT_TYPE_GUID	A GUID value

You'll be setting and retrieving attributes as byte arrays, which usually means that you have to explicitly cast your variables. The code examples at the end of this document demonstrate how to work with the data types.

Reading Metadata

After you have a file loaded into the metadata editor object, you can retrieve attributes using the methods of the **IWMHeaderInfo3** interface. The basic process for reading metadata is to define your search criteria and get a list of matching attribute indices by using the **GetAttributeIndices** method. Once you have the matching indices, you use the **GetAttributeByIndexEx** method to retrieve the name/value pair of each attribute.

The following topics describe this process in detail.

Getting Attribute Indices

The **GetAttributeIndices** method provides a flexible way to find the attributes in a file. Here is the signature of the method:

```
HRESULT GetAttributeIndices(
   WORD wStreamNum,
   LPCWSTR pwszName,
   WORD *pwLangIndex,
   WORD *pwIndices,
   WORD *pwCount
);
```

The stream number is always required. You can use zero for file-level attributes (most content-descriptive metadata is file-level). You can also use 0xFFFF to get indices for all matching attributes in the file regardless of stream number.

You can search on the attribute name, the language index, or both. If you want to search on only one, the other argument must be set to NULL. Language indices are outside the scope of this document, so all code shown here uses language index 0 (the default language on the user's machine). See the Format SDK documentation for more information. If you search by name, be certain that the name string you pass is NULL-terminated. If using an attribute name defined in the Format SDK, you should always use the global constant. You should make your own constants for custom attributes too, so that typographical errors will be caught at compile time.

You need to make two calls to **GetAttributeIndices** for every search. On the first call, pass NULL for the indices array. If successful, the method will set the value referenced by *pwCount* to the number of elements in the indices array. Then you can allocate memory to hold the indices and call a second time to get them.

The following code shows how to get the indices for all instances of the g_wszWMTitle attribute (there is typically only one, though there may be differences in title by language that necessitate multiples):

```
IWMHeaderInfo3* pHeaderInfo = NULL; // We'll assume you'll query
                                    //  for this before the code
                                    //  following.

WORD* pwIndices = NULL; // Will reference the dynamically
                        //  allocated array pointer.
WORD  wNumIndices = 0; // Number of indices.

HRESULT hr = S_OK;
.
.
.
// Get the number of indices
hr = pHeaderInfo->GetAttributeIndices(
           0xFFFF, // Special stream number for all instances.
           g_wszWMTitle, // Constant defined in SDK
           NULL, // We're not worried about language.
           NULL, // No array yet, just getting the size.
           &wNumIndices);
if(wNumIndices < 1){
   // No matching indices. Exit out of whatever logical unit
   //  we're in.
}

// Allocate the array.
pwIndices = new WORD[wNumIndices];

if(pwIndices == NULL){
   // Out of memory. Do whatever is appropriate to your application.
}

// Now get the indices.
```

```
hr = pHeaderInfo->GetAttributeIndices(
          0xFFFF,
          g_wszWMTitle,
          NULL,
          pwIndices,
          wNumIndices);

// Code for getting each attribute instance goes here.

// Don't forget to release the array when finished!
if(pwIndices != NULL){
   delete[] pwIndices;
   pwIndices = NULL;
}
```

If you want to retrieve all of the metadata in the file (or in a particular stream) you can use the much simpler **GetAttributeCount**, which takes the stream number and sets the count value on return.

Getting Attributes

Getting attributes with **GetAttributeByIndexEx** is similar to getting the indices. You have to make two calls, the first to get the sizes of the arrays for name and data and the second to put that data into arrays you allocate. Here is the signature for the method:

```
HRESULT GetAttributeByIndexEx(
   WORD wStreamNum,
   WORD wIndex,
   LPWSTR pwszName,
   WORD *pwNameLen,
   WMT_ATTR_DATATYPE *pType,
   WORD *pwLangIndex,
   BYTE *pValue,
   DWORD *pdwDataLength
);
```

The stream number, index, and language index are the same as in **GetAttributeIndices**, though language index is an output value for this method.

As you can see, the method provides the value of the attribute as an array of bytes. To help you handle retrieved metadata, it also gives you the data type, as a value in the **WMT_ATTR_DATATYPE** enumerated type. You can safely cast the attribute value to the type specified.

The following snippet shows how to get an attribute by its index:

```
WORD   wIndex = NULL; // We'll assume you use GetAttributeIndices
                      //  to get this before the meat of the following
                      //  code.
WCHAR* pwszName = NULL;
WORD   cchName  = 0;
WORD   Language = 0;
BYTE*  pbValue  = NULL;
DWORD  cbValue  = 0;

WMT_ATTR_DATATYPE AttType;

// Find the lengths of the attribute name and value.
hr = pHeaderInfo->GetAttributeByIndexEx(
          0,
          wIndex,
          NULL,
          &cchName,
          NULL,
          NULL,
          NULL,
          &cbValue);

// Allocate memory for the name and value.
pwszName = new WCHAR[cchName];
pbValue  = new BYTE[cbValue];

if(pwszName == NULL || pbValue == NULL)
{
   // Out of memory. Do whatever is appropriate.
}
```

```
// Get the attribute.
hr = pHeaderInfo->GetAttributeByIndexEx(
          0,
          wIndex,
          pwszName,
          &cchName,
          &AttType,
          &Language,
          pbValue,
          &cbValue);

// Don't forget to release dynamically allocated memory!
```

Writing and Editing Metadata

You can set metadata attributes in a file by using the **AddAttribute** method. This method call is very similar to **GetAttrbitueByIndexEx**, as you can see in its signature:

```
HRESULT AddAttribute(
   WORD wStreamNum,
   LPCWSTR pszName,
   WORD *pwIndex,
   WMT_ATTR_DATATYPE Type,
   WORD wLangIndex,
   const BYTE *pValue,
   DWORD dwLength
);
```

Because you are passing in the name and value, you only need to make this call once. Note that the attribute name string does not have a corresponding length argument, so it is essential that you ensure the string is NULL-terminated.

The following code shows how to set a custom attribute:

```
HRESULT hr = S_OK;
LPCWSTR pszAttributeName = L"JLRating";
WORD wIndex = 0;
// I'm assuming here that I want a five-star rating system for
//  files in my application. I want to start them in the middle,
```

```
//  at three stars as a default. Different applications use
//  rating in different ways, which is why I'm using a custom
//  attribute.
WORD wRating = 3;
.
.
.
hr = pHeaderInfo->AddAttribute(
          0, // Stream 0 for file-level attribute
          pszAttributeName,
          &wIndex, // Will contain the index on return
          WMT_TYPE_WORD,
          0, // Default language (en-us for me)
          (BYTE*)&wRating,
          sizeof(wRating));
```

Because multiple copies of a given attribute can exist in a file, it is good practice to check for the existence of an attribute before adding a new one.

Be aware that the index provided on return of the method is the index for the stream number provided. You must use a stream number, with stream 0 indicating a file-level attribute—you cannot use stream 0xFFFF to set an attribute.

You can use the **ModifyAttribute** method to edit an attribute that already exists in the file. That method call works identically to **AddAttribute**, except that you indicate the attribute by index instead of by name.

Deleting Metadata

You can delete metadata attributes by using the **DeleteAttribute** method. It takes a stream number and the index of the attribute to delete.

Deleting an attribute does not change the order of attribute indices, but it does decrement all of the attribute indices of greater value. This can be inconvenient if you need to delete multiple attributes. To assist in attribute deletion, the **GetAttributeIndices** method sets the indices array in descending order. You can loop through the indices deleting attributes as needed as you iterate through them without the resulting changes in indices affecting your operations.

Working with "Complex" Metadata Attributes

Some attributes use the **WMT_TYPE_BINARY** enumeration as their data type. These usually have values that are represented by structures, in which case they are called complex attributes. That name is perhaps misleading, as there are no advanced techniques required to deal with them. Rather, it is an

indication that you should retrieve the value into the appropriate data structure. The second example in the code examples section of this documentation demonstrates this process.

Code Examples

The two code examples in the following sections provide more complete demonstrations of metadata operations than the snippets in the rest of this document. Each has been structured as one or more functions to which you pass pointers to the relevant interfaces and other data.

Macros Used in the Code Examples

These examples use some macros that are common to the Format SDK documentation.

The first is GOTO_EXIT_IF_FAILED, which diverts execution to an error handling block if it finds that the specified HRESULT indicates a failed COM call. This method of error handling relies on a label named Exit, which indicates the start of error handling at the end of a function. This use of the otherwise discouraged goto statement is well known in the Format SDK. Here is the code for the macro:

```
#ifndef GOTO_EXIT_IF_FAILED
#define GOTO_EXIT_IF_FAILED(hr) if(FAILED(hr)) goto Exit;
#endif
```

The other two macros are SAFE_RELEASE, and SAFE_ARRAY_DELETE, both of which check to see if a pointer (to an interface and an array respectively) is NULL. If the pointer is not null, the macros release the reference and set the pointer to NULL. These are used in the error handling blocks at the end of functions, which could be entered in a variety of states, depending on whether any of the calls returned errors. Here is the code for both macros:

```
#ifndef SAFE_RELEASE
#define SAFE_RELEASE(x) \
   if(x != NULL){       \
      x->Release();     \
      x = NULL;         \
   }
#endif

#ifndef SAFE_ARRAY_DELETE
#define SAFE_ARRAY_DELETE(x) \
   if(x != NULL){            \
      delete[] x;            \
      x = NULL;              \
   }
```

```
#endif
```

Retrieving All Metadata in a File

This example function, **ShowAllAttributes**, takes as its only parameter a pointer to the **IWMHeaderInfo3** interface of an object in which a file has already been opened. It gets a count of all the attributes in the file and then gets each one and prints it to the console.

The values of attributes of the **WMT_TYPE_BINARY** and **WMT_TYPE_GUID** types are not displayed, as the code required to navigate these structures would only complicate the example.

```
HRESULT ShowAllAttributes(IWMHeaderInfo3* pHeaderInfo){
    HRESULT hr          = S_OK;

    WORD    cAttributes = 0;
    WCHAR*  pwszName    = NULL;
    WORD    cchName     = 0;
    BYTE*   pbValue     = NULL;
    DWORD   cbValue     = 0;
    WORD    langIndex   = 0;
    WORD    attIndex    = 0;

    WMT_ATTR_DATATYPE attType;

    // Get the total number of attributes in the file.
    hr = pHeaderInfo->GetAttributeCountEx(0xFFFF, &cAttributes);
    GOTO_EXIT_IF_FAILED(hr);

    // Loop through all the attributes, retrieving and
    //  displaying each.
    for(attIndex = 0; attIndex < cAttributes; attIndex++){
        // Get the required buffer lengths for the name and
        //  value.
        hr = pHeaderInfo->GetAttributeByIndexEx(
                  0xFFFF,
                  attIndex,
                  NULL,
                  &cchName,
```

```
                NULL,
                NULL,
                NULL,
                &cbValue);
        GOTO_EXIT_IF_FAILED(hr);

        // Allocate the buffers.
        pwszName = new WCHAR[cchName];
        if(pwszName == NULL){
            hr = E_OUTOFMEMORY;
            goto Exit;
        }

        pbValue = new BYTE[cbValue];
        if(pbValue == NULL){
            hr = E_OUTOFMEMORY;
            goto Exit;
        }

        // Get the attribute.
        hr = pHeaderInfo->GetAttributeByIndexEx(
                0xFFFF,
                attIndex,
                pwszName,
                &cchName,
                &attType,
                &langIndex,
                pbValue,
                &cbValue);
        GOTO_EXIT_IF_FAILED(hr);

        // Display the attribute global index and name.
        printf("%3d - %S (Language %d):\n\t ", attIndex,
               pwszName, langIndex);
```

```
        // Display the attribute depending upon type.
        switch(attType){
        case WMT_TYPE_DWORD:
            printf("%d\n\n", (DWORD)*pbValue);
        case WMT_TYPE_QWORD:
            printf("%d\n\n", (QWORD)*pbValue);
        case WMT_TYPE_WORD:
            printf("%d\n\n", (WORD) *pbValue);
            break;
        case WMT_TYPE_STRING:
            printf("%S\n\n", (WCHAR*) pbValue);
            break;
        case WMT_TYPE_BINARY:
            printf("<binary value>\n\n");
            break;
        case WMT_TYPE_BOOL:
            printf("%s\n\n",
                 ((BOOL) *pbValue == TRUE) ? "True" : "False");
            break;
        case WMT_TYPE_GUID:
            printf("<GUID value>\n\n", (DWORD) *pbValue);
            break;
        } // End switch

        // Release allocated memory for the next pass.
        SAFE_ARRAY_DELETE(pwszName);
        SAFE_ARRAY_DELETE(pbValue);
        cchName = 0;
        cbValue = 0;
    } // End for attIndex.

Exit:
    SAFE_ARRAY_DELETE(pwszName);
```

```
    SAFE_ARRAY_DELETE(pbValue);
    return hr;
}
```

Using Complex Metadata Attributes

This example consists of two functions to demonstrate the use of the **WM/Text** complex metadata attribute. **WM/Text** (gwszWMText) is an attribute that holds a pair of strings: user text and a description of what that text signifies.

The first function, **AddText**, adds a **WM/Text** attribute to the file. Its arguments include a pointer to the **IWMHeaderInfo3** interface of an object into which a file has been opened, the description and text strings for the attribute, and a pointer to a WORD value which the function will set to the index of the new attribute once added.

The second function, **DisplayText**, takes the interface pointer and the index number returned by **AddText** and uses them to retrieve the attribute and print it to the console.

```
HRESULT AddText(IWMHeaderInfo3* pHeaderInfo,
                WCHAR* pwszDesc,
                WCHAR* pwszText,
                WORD* pwIndex){
    HRESULT hr         = S_OK;
    WORD    wIndex     = 0;

    WM_USER_TEXT textStruct;

    // Populate the text structure.
    textStruct.pwszDescription = pwszDesc;
    textStruct.pwszText = pwszText;

    // Add the attribute.
    hr = pHeaderInfo->AddAttribute(
              0,
              g_wszWMText,
              &wIndex,
              WMT_TYPE_BINARY,
              0,
              (BYTE*)&textStruct,
```

```
                sizeof(WM_USER_TEXT));

    // Pass the index of the text attribute back to the caller.
    if(SUCCEEDED(hr))
    {
        *pwIndex = wIndex;
    }

    return hr;
}

HRESULT DisplayText(IWMHeaderInfo3* pHeaderInfo, WORD wIndex){
    HRESULT hr = S_OK;

    WCHAR*  pwszName = NULL;
    WORD    cchName  = 0;
    WORD    Language = 0;
    BYTE*   pbValue  = NULL;
    DWORD   cbValue  = 0;


    WM_USER_TEXT*     pText    = NULL;
    WMT_ATTR_DATATYPE AttType;


    // Find the lengths of the attribute name and value.
    hr = pHeaderInfo->GetAttributeByIndexEx(
              0,
              wIndex,
              NULL,
              &cchName,
              NULL,
              NULL,
              NULL,
```

```
            &cbValue);
    GOTO_EXIT_IF_FAILED(hr);

    // Allocate memory for the name and value.
    pwszName = new WCHAR[cchName];
    pbValue  = new BYTE[cbValue];

    if(pwszName == NULL || pbValue == NULL){
        hr = E_OUTOFMEMORY;
        goto Exit;
    }

    // Get the attribute.
    hr = pHeaderInfo->GetAttributeByIndexEx(
              0,
              wIndex,
              pwszName,
              &cchName,
              &AttType,
              &Language,
              pbValue,
              &cbValue);
    GOTO_EXIT_IF_FAILED(hr);

    // Make sure the attribute is WM/Text, as expected.
    if(wcscmp(pwszName, g_wszWMText)){
        // Somehow we got the wrong attribute.
        hr = E_UNEXPECTED;
        goto Exit;
    }

    // Set the structure pointer to the retrieved value.
    pText = (WM_USER_TEXT*) pbValue;
```

```
    // Print the strings from the structure.
    printf("Description : %S\n", pText->pwszDescription);
    printf("Text        : %S\n", pText->pwszText);

Exit:

SAFE_ARRAY_DELETE(pwszName);
    SAFE_ARRAY_DELETE(pbValue);
    return hr;
}
```

Conclusion

The ASF file format, in conjunction with the functionality provided by the objects of the Windows Media Format SDK, gives you a flexible and powerful metadata solution for your media applications. This document has shown you the basic functionality of the APIs in the SDK. The real power of metadata in ASF files comes from the unique and creative solutions that you implement in your applications.

Please refer to the Windows Media Format SDK for further information.