AwCOM 3
The Awave Streaming Audio Converter
COM library

Reference manual

Table of Contents:

Introduction:

The Awave Streaming Audio Converter COM library v3.x from FMJ-Software, or AwCOM 3 for short, is a COM dual interface, in-process DLL server component. It provides functions for converting between several different audio waveform file formats. It can also optionally perform several forms of processing on the audio data. The library is broken up into a modular set of "filters" that are connected in a "graph" through which the audio data is streamed. Most modules are licensed separately – so you only have to pay for the parts that you use.

AwCOM 3 is built as a wrapper on top of AwC++ – a C++ class library from FMJ-Software intended for doing various audio streaming & conversion tasks. The Awave Audio, ACDR and Chromatia Tuner software from FMJ-Software are examples of commercial products built using AwC++.

The file License.rtf contains the details of your licensing agreement for AwCOM 3.

/ Markus of FMJ-Software,   http://www.fmjsoft.com/

Symbol index

Usage section:

The AwCOM 3 component, implemented by AwCOM3.dll, consists of two object classes: AwManager, which expose the IAwManager interface, and AwGraph that exports the IAwGraph interface. The AwManager class contains functions for managing your 'license keys', for enumerating available file readers and file writers, for creating AwGraph objects, and a few miscellaneous functions. The AwGraph class represents a 'graph' of filters that controls the 'flow' of the audio data during a conversion – and through which the audio data is 'streamed'.

File organization

The files compromising the library are organized as follows:

directory:	what:
Doc	Documentation
Bin.x86	32-bit versions of the AwCOM run-time
Bin.x64	64-bit versions of the AwCOM run-time
C++	Resources for using AwCOM with C++
Delphi	Resources for using AwCOM with Borland Delphi 3 or later
VB	Resources for using AwCOM with Visual Basic 5 or later

The README.TXT file in the main directory contains a more complete file listing.

Installing the SDK

To install the AwCOM 3 SDK environment, simply unpack all of the files from the distribution archive.

To install the AwCOM3 run-time on your machine, run the AwInst3.exe program (found in Bin.*).

Note: When running under Windows Vista or later, you must allow AwInst3.exe to run with administrator privileges (the UAC should pop up and ask for it if necessary – the files are installed to ...\<program files>\Common Files\System\AwCOM – a location with restricted write access).

Redistributing AwCOM 3

The easiest way by far to install or uninstall AwCOM 3 on a machine is to run the `AwInst3.exe` program (uninstall by passing it an "`-u`" command line switch). This installs (or uninstalls) the `AwCOM3.dll`, plus any external codec dll's required. It is recommended that you use `AwInst3.exe` when redistributing AwCOM 3 with your application - but if you wish, you may use a modified version of `AwInst3` (the C++ source code is supplied in the `C++/AwInst3` directory), or you may opt to use an installation program of your own.

Files to include

`AwInst3.exe`	Optional installation program
`AwCOM3.dll`	The AwCOM in-process server.
`<*.dll>`	Any additional codec DLL's that may be required.

Custom run-time installation procedure

If you choose to provide your own installation/unistallation program, other than AwInst3, then you must be careful to observe the following 'rules'. (Note: many specialized installation creation tools (e.g. Install Shield) will handle many of these things for you automatically, but you still should be aware of them).

- When installing a DLL file you must always first check if it is already present at the intended location. If it is then you must compare the file version numbers stored in the 'fixed file info' section of it's file resource part. You must never overwrite a file with a newer version number with a file with an older version number. Newer files are guaranteed to be backward compatible. The file version numbers can be retrieved using the standard Windows 'File Installation Library' API.

- When installing a DLL, increment its 'usage count' in the Windows registry. When uninstalling, decrement its usage count. Never delete any file before it's usage count has reached zero (or less). The 'usage count' is stored in the Windows registry under "`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs`" as an entry with the full DLL file name and path as the entry name, and the usage count as the entry data (stored as `DWORD` type).

- Get the AwCOM 'base path' - usually "`C:\Program Files\CommonFiles\System\AwCOM`" - but this could vary on international language versions of Windows. To construct the base path, fetch the value of the Windows registry entry "`CommonFilesDir`" under "`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`". Then append "`\System\AwCOM`" to complete the path.

- Install/uninstall `AwCOM3.DLL` in the 'base path' (see above). The location is very important because the AwCOM 3 class may be shared between applications from multiple vendors and there must be only one instance of it installed on a system.

- Register the AwCOM 3 classes. This should be done by letting it 'self register' by running '`RegSvr32.exe "<base path>\AwCOM3.DLL"`' (add the -u switch before the path if you want to unregister it). Note that RegSvr32 is a standard Windows component (always be present in the Windows system directory).

- Install/uninstall any additional codec DLL's that may be required for the file formats that you are going to use (e.g. `Lame_Enc.dll` for writing MP3 files).

GUIDs and interfaces

GUID's (a unique 128-bit number) are used to identify interfaces and other objects by COM. On top of COM there exists a 'dynamically typed component environment' often referred to as the 'dispatch interface' and required by some languages such as VisualBasic. Dispatch enabled components can be referred to by name instead of by GUID. AwCOM 3 can be called both using the 'native' COM interfaces and the less effective (but much more user friendly) dispatch interface.

The GUID's identifying the various exposed parts AwCOM 3 are as follows:

| Object	GUID	Dispatch name
AwCOM3 v1.0 type library	64EF5CB3-DAC4-4EBD-855C-2F41A69D4290	`AwCOM3`
AwManager class id	AF0C2EC1-9715-4A2E-A42F-0A148910FA4C	`AwCOM3.AwManager`
AwGraph class id	0B2E60B4-0DBD-41B2-97B9-779757BA20BC	`AwCOM3.AwGraph`
IAwManager interface id	15D81CE2-68C1-4F5D-82B8-F645B81EB18B	n/a
IAwGraph interface id	F2B0A306-2A93-432D-9D25-02A0ADC5CD00	n/a
Error codes	n/a	`AwCOM3.AwErrors`

A note on strings

If you are using Visual Basic, Delphi or some similar environment that can use 'OLE strings' natively and/or transparently, then you can skip this section!

In order to be compatible with 'dispatch' based clients, AwCOM 3 uses 'standard OLE Automation strings'. Some environments (e.g. Visual Basic) handles these strings transparently – so you don't have to worry about this at all, while in others (e.g. Visual C++) you must use `SysAllocString`, `SysFreeString` and related Windows API functions to manage the strings (Visual C++ provides a _bstr_t class to help with this). An important thing to know if you are managing the strings yourself, is the convention that input string parameters (BSTR) are both allocated and later freed by the client (your program), while "by reference" return strings (BSTR *) are allocated by the server (AwCOM3) and freed by the client (your program).

Filter graphs

The AwCOM 3 library implements audio data processing as a filter graph. The graph contains a simple chain of filters connected in series. The audio is streamed in small packets of data from one end of the graph-chain to the other, as illustrated in the following figure:

E.g: File reader   Resampling   File writer

⟶ Flow of data packets
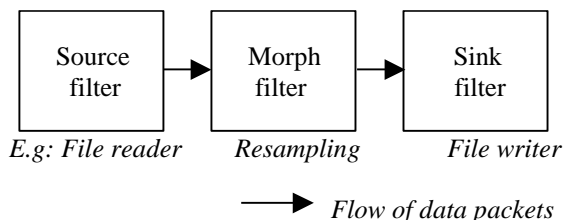
A source filter is any filter that has an output, while a sink filter has only an input, and a morph filter has both. Morph filters are used to process, monitor or just rearrange data. You can have any number of morph filters but there must only be one source and one sink.

The conversion process

The details of using AwCOM 3 when you get down to earth is dependent on the language that you use – therefore it is detailed for a few different languages in subsequent sections. Here however, is given a general outline to familiarize you with the concepts involved.

Initialization

You start by creating an `AwManager` object and get it's `IAwManager` interface.

Next you have to call `AwManager.SetProperty("License", params…)` one or more times in order to enable the parts of AwCOM 3 for which you have bought licenses. For your particular set-up this must be done as follows:

```
AwManager.SetProperty("License", "name=ANameYouGotFromUs")
AwManager.SetProperty("License", "wav=AKeyYouGotFromUs")
AwManager.SetProperty("License", "mpeg=AnotherKeyYouGotFromUs")
```

Next call `AwManager.CreateEasyGraph` or `AwManager.CreateGraph` to create an `AwGraph` object and obtain its `IAwGraph` interface.

Easy graph setup – CreateEasyGraph

If you don't want any in-depth control of the filter graph setup – maybe you just want to convert from one file format to another with as little fuss as possible – then use the `AwManager.CreateEasyGraph` function to create a pre-built graph. Here's an example that converts a WAV file to an 8000 Hz, 1-channel AU file with mu-Law data:

```
AwGraph = AwManager.CreateEasyGraph(
        "input.wav",
        "output.au",
        -1,
        "mu-Law",
        8000,
        1,
        &pGraph)
```

The first parameter is the input file name. The second is the output file name. The third is an enumeration identifier for the output file format (see `AwManager.EnumFileWriters`) – use the value -1 to auto-detect from the file extension. The fourth is the desired output data format (see `AwManager.EnumFileWriterFormats`). The fifth is the desired sample-rate – use -1 to keep the input sample rate. The sixth and last is the number of output channels – use -1 to use the same number of channels as the input data (if possible, if not, the best nearest channel count it used).

Manual graph setup – CreateGraph

If you want more control over what's happening – then use the `AwManager.CreateGraph` function to create an empty graph:

```
AwGraph = AwManager.CreateGraph(&pGraph)
```

Then manually add whatever filters you want – in the exact sequential order that you want them and with the parameters of your choice. Here's an example that sets up the same graph as in the EasyGraph example:

```
AwGraph.AddFilter("FileReader", "file=\"input.wav\"")
AwGraph.AddFilter("Resampler", "samplerate=8000")
AwGraph.AddFilter("ChannelBroker", "outchannels=1")
AwGraph.AddFilter("FileWriter",
                  "file=\"output.au\" dataformat=\"mu-Law\"")
```

After adding the `FileReader` filter, you can start querying the graph about input file properties (see `AwGraph.GetPropery`) such as sample rate, length and number of channels. You can also retrieve and/or set various informational properties such as author, copyright, comment, track number, loop point, et c.
After adding a `FileWriter` filter, you can query output file info (sample rate, channel count, and length).
Note that when adding a reader or a writer, it will immediately require access to the files – they mustn't be locked by another app or the call will fail.

Execution

Finally – to perform the actual conversion process – i.e. to start streaming the data through the graph - simply call `AwGraph.Execute`. The call will not return until the conversion is either finished, or an error occurred (in which case it is aborted). Note that you can only call it only once, calling it a second time has undefined results.

NB: The AwCOM 3 library's threading model is "both", which means that it can work both with the "apartment threading" and the "free threading" (a.k.a. "multithreading") models. It is recommended that you keep only one single instance of `AwManager` and use it to create `AwGraph`'s. If you want to run concurrent threads, then the graphs can be handed over to dedicated worker threads.

Error handling

Although omitted in the examples above – don't forget to check the error codes returned by all the function calls! They all return a `HRESULT` value, however some languages, e.g. Delphi and VisualBasic, hides this and instead raise an exception when an error occurs, in which case you need to add an exception handler to catch any errors. To get a text description from an error code, use:

```
AwManager.DescribeError(lErrorCode, &strDescription).
```

Progress reporting

There is no COM-based notification interface for reporting the progress of the executing graph (the progress of the file conversion). However, there is a hack to obtain the progress status:

– Before executing the graph, you may call:

```
AwGraph.SetProperty("ProgressCallback", strProcHexAddress).
```

… where `strHexAddress` is a string containing the memory address of a callback-function written – encoded as an 8/16-digit Hexadecimal number. The callback function will be called periodically during the graph execution. The function must use the Windows standard calling convention (`__stdcall` in C++), taking a single integer as parameter (the progress state in percent), and having no return value.

Here's an example for 32-bit VB.NET using a delegate as a callback (a little bit tricky…)

```
Public Sub CallBackFunc(ByVal PercentValue As Int32)
      … handle progress callback here …
End Sub

Delegate Sub DelegateProgress(ByVal PercentValue As Int32)

Private Sub SetAwProgress(ByRef tsMsgError As String)
      Dim myDelegate As New DelegateProgress(AddressOf CallbackFunc)
      Dim myPtr As IntPtr = Marshal.GetFunctionPointerForDelegate(myDelegate)
      Dim hexAddress As String = Hex(myPtr.ToInt32)
      Call moAwGraph.SetProperty("ProgressCallback", hexAddress)
End Sub
```

Freeing file handles

The graph may hold file handles open for the input and/or output file until the time that the graph object is released. The object is normally released when you call `AwGraph.Release()`. However, some languages (e.g. VB) which use garbage collection for memory management will hide this function from the programmer – making it difficult to determinate exactly when a file handle will be released – preventing a newly written file from being opened in another app. In this case you can explicitly release the file handles by calling:

```
AwGraph.SetProperty("ClearGraph", "true").
```

Using AwCOM 3 with C++

For C++ users, a header file, AwCOM3.h, with AwCOM 3 definitions has been provided in the C++ subdirectory.

The GUID variables that you need - `CLSID_AwManager`, `IID_IAwManager` and `IID_IAwGraph` - are per default declared as 'extern'. If you place `#define INITGUID` statement before including the header file, they will instead be compiled as normal variables. I.e. in order to compile and link them once and once only, you must use the INITGUID declaration (before including the header file) once and once only in each project.

MFC users note: MFC users should use `#include "initguid.h"` instead of `#define INITGUID`.

The IDL interface definition has been complemented with more default parameter values for C++ use. E.g. many pointers to return values are `NULL` by default (in this case no value is returned for that parameter).

If you use Visual C++ v6 or later, you can use the `_bstr_t` helper class to manage string operations (`#include <comdef.h>` to get the `_bstr_t` class definitions). Use `_bstr_t(<BSTR>, false)` to 'take over' a string returned by an AwCOM 3 function. Use `_bstr_t(<char *>)` to create a BSTR to send as input to an AwCOM 3 function. The `_bstr_t` class also contains functions for easily converting to and from normal 8-bit ASCII (simply type cast it to `(const char *)`) or 16-bit UNICODE string (type cast it to `(const WCHAR *)`) and BSTR's. For more information see the Visual C++ help files.

C++ Examples

In the `C++/AwConv` directory is a C++ program called `AwConv.cpp`. The supplied project files are for Visual C++ 2010 and later. It is a simple command line audio file format converter. It should be set up and compiled as a console mode application. When compiled, simply run `AwConv.exe` to get a list of command line options. The source code contains some useful hints about how to use the AwCOM 3 object.

Here's also a short example of how to open a .WAV file, resample it to 8000 Hz, and then save it as a mono, mu-law data-type of the AU file format. Note that error handling is omitted for increased readability.

```
// Instantiate AwManager
IAwManager *pawMan = NULL;
CoCreateInstance(CLSID_AwManager, NULL, CLSCTX_INPROC_SERVER,
                 IID_IAwManager, (LPVOID *)&pawMan);

// Set license
pawMan->SetProperty(_bstr_t("License"),_bstr_t("name=\"Your license name here\""));
pawMan->SetProperty(_bstr_t("License"),_bstr_t("wav=Wav license key here"));
… et c …

// Create a graph
IAwGraph *pawGraph = NULL;
pawMan->CreateGraph(&pawGraph);

// Add filters…
pawGraph->AddFilter(_bstr_t("FileReader"), _bstr_t("file=\"input.wav\""));
pawGraph->AddFilter(_bstr_t("Resampler"), _bstr_t("samplerate=8000"));
pawGraph->AddFilter(_bstr_t("ChannelBroker"), _bstr_t("outchannels=1"));
pawGraph->AddFilter(_bstr_t("FileWriter"),
                    _bstr_t("file=\"out.au\" dataformat=\"mu-Law\""));

// Execute graph!
pawGraph->Execute()

// Clean up
pawGraph->Release();
pawMan->Release();
```

Using AwCOM 3 with Delphi

Borland Delphi can use both dispatch and v-table based COM objects. As it is the more effective method, only the v-table approach will be described here. As a convenience, the source code for a 'Delphi unit' file, AwCOM3.pas, with the necessary definitions for using AwCOM 3 have been provided in the Delphi subdirectory.

Delphi's `WideString` type corresponds to the OLE string type (`BSTR`) so if you use that, you should not have to worry about string memory management. Delphi will also in most cases automatically translate between its different 'native' string types!

Because all the methods and properties are defined as `safecall`, Delphi will hide the `HRESULT` return from them. Instead all methods with the [retval] flag set for one parameter in the IDL definition will be seen as functions (with the return value specified by the retval parameter), and the rest as methods. Whenever a method returns an error code, an `EOleException` exception will be issued. The AwCOM 3 error code (from the hidden `HRESULT` return) is found in `EOleException.ErrorCode`.

Add `uses AwCOM3` to your program to get the interface declarations as well as a few helper functions.

To instantiate AwCOM 3 and retrieve the `IAwManager` interface, you can use the helper function:

```
CoAwCOM3.Create
```

Delphi Example

Here's a short example of how to open a .WAV file, resample it to 8000 Hz mono mu-law AU file:

```
uses AwCOM3, SysUtils, ComObj;

var
awMan: IAwManager;
awG: IAwGraph
begin
// Instantiate an AwManager object
awMan := CoAwCOM3.Create;

try
      // Set license keys
      awMan.SetProperty('License', 'name="MyCompany" wav=… au=… et c");

      // Create a new graph
      awG = awMan. CreateGraph;

      // Add filters
      awG.AddFilter('FileReader', 'file="input.wav"');
      awG.AddFilter('Resampler', 'samplerate=8000');
      awG.AddFilter('ChannelBroker, 'outchannels=1');
      awG.AddFilter('FileWriter', 'file="output.au" dataformat="mu-Law"');

      // Run it!
      awG.Execute

except // Catch any AwCOM 3 errors

      on E: EOleException do
            DispError('Error: ' + awMan.DescribeError(E.ErrorCode));
      end;
end.
```

Using AwCOM 3 with Visual Basic

The AwCOM3.DLL server provides a full type library, which makes it very easy to use it from Visual Basic. The first thing you have to do is to tell Visual Basic where to find this type library.

To add the AwCOM component to your VB environment:
> Select Project -> References… -> Browse…
> Go to `\Program Files\Common Files\System\AwCOM`, select `AwCOM3.DLL` and click Open.
> Check the box beside '`AwCOM3 1.0 Type Library`' that should now be in the 'references' list.

The Object Browser

Visual Basic will format the methods and properties a little different than how the `IAwManager` and `IAwGraph` reference sections in this document defines them (in IDL syntax). E.g. all methods with a `retval` attribute will be seen as functions instead of as methods. The `HRESULT` return codes will not be seen but will be raised as errors in case a function does not return a success code. Because of these differences, it's a good idea to use the object browser as a reference to the AwCOM 3 class syntax in Visual Basic. All functions and declarations will have a short help text defined in the Object Browser (look under `AwManager`, `AwGraph`, and `AwErrors`). When you need more explicit documentation for the parameters, look it up in this document.

VB Examples

There are two Visual Basic 6 sample applications included:
- `VBSimple` demonstrates the easiest way to do conversions (using a single `CreateEasyGraph` call).
- `VBConv` is a bit more complex and demonstrates how to manually construct conversion graphs.

The following sample demonstrates how to open a .WAV file and save it as 8000 Hz mono, mu-law format .AU:

```
' Instantiate objects
On Error GoTo AwErrHandler
Dim awG As AwGraph
Dim awMan As AwManager
Set aw = New AwCOM
Set awG = awMan.CreateGraph

' Create filter graph
awG.AddFilter "FileReader", "file=input.wav"
awG.AddFilter "Resampler", "samplerate=8000"
awG.AddFilter "ChannelBroker", "outchannels=1"
awG.AddFilter "FileWriter", "file=output.au dataformat=""mu-Law"""

' Do it!
awG.Execute
GoTo Done

AwErrHandler:
MsgBox "Error: " + aw.DescribeError(Err.Number)

Done:
' Free objects
Set awG = Nothing
Set awMan = Nothing
```

NB: To set a progress callback function in VB6 you must create a .bas module with a public function declared like thus: `Public Function ProgressCB (Byval mPercentValue as integer) as long`
Then call: `AwGraph.SetProperty("ProgressCallback", HEX(AddressOf ProgressCB)))`

Reference section

The COM standard 'IDL definitions' are given in this section.
The actual syntax used by a particular language is somewhat language dependent.

Error codes

All functions return an HRESULT value used to indicate either success or failure if some kind.

Success:

```
Value       Name              Description
0           awOk              Function call succeeded.
1           awFalse           Function call succeeded but nothing to return.
```

Failure:

```
Value       Name              Description
80040200h   awGeneral         A general unspecified error occurred.
80040201h   awInvalidParam    An invalid parameter was supplied.
80040202h   awOutOfMemory     Out of memory while performing operation.
80040203h   awFileOpen        Could not open the specified file.
80040204h   awFileCreate      Could not create the specified file.
80040205h   awFileRead        Could not read from the input file.
80040206h   awFileWrite       Could not write to the output file.
80040207h   awInvalidOpSequence An invalid sequence of calls were made.
80040208h   awUnsupDataFormat Unsupported data format was encountered.
80040209h   awUnsupChannelFormat Unsupported channel format encountered/specified.
8004020Ah   awUnsupFileFormat Input file was not of any recognized file format.
8004020Bh   awUnsupCompression Unsupported compression format in the input file.
8004020Ch   awInvalidFileType An invalid file type was encountered.
8004020Dh   awCorruptedFile   The input file was corrupted and reading failed.
8004020Eh   awEmptyFile       The input file did not contain any waveform data.
8004020Fh   awInvalidLicense  No valid AwCOM license could be found.
80040210h   awMissingComponent Could not locate an external codec DLL.
80040211h   awCancelled       Operation cancelled.
80040212h   awNoRights        No read-rights - copy protected file.
```

Note: All the above values are given in hexadecimal number base notation.

The IAwManager interface

The `IAwGraph` is the main interface to an `AwManager` object containing the top level object in an AwCOM 3 server. All functions return an `HRESULT` value.
The actual syntax used by a particular language is somewhat language dependent. E.g. parameters marked with `[retval]` will be returned as function results in languages that hides the COM HRESULT values.

As all COM interfaces, it inherits from `IUnknown` and thus have `QueryInterface, AddRef` and `Release` methods – it is assumed that you are familiar with these – they are not documented here (note that some languages, e.g. VisualBasic, will hide these methods and handle all the reference counting for you).

```
HRESULT GetProperty(       [in] BSTR strName,
                           [out, retval] BSTR *pstrValue );
```

Get an informational property.

Input:

    `strName`   Name of property to get.

Output:

    `*pstrValue`   Property value (And empty string if property not available).

The following properties can be retrieved:

Property name:	Property value description:
Version	AwCOM 3 library version.
Name	Liberay name.
Copyright	Library copyright string.
Licensee	Name of licensee (your company name).
QuantizationType	Bit depth quantization method.

```
HRESULT SetProperty(       [in] BSTR strName,
                           [in] BSTR pstrValue );
```

Set an informational property.

Input:
      `strName`   Name of property to set.
Output:
      `pstrValue`   Property value.

The following properties can be set:

Property name:	Property value description:
`License`	Set license key – see further below.
`QuantizationType`	Select the bit depth quantization method (default=`"RoundNearest"`): Available types: `"RoundNearest"`, `"RectWhiteNoise"`, `"TriWhiteNoise"`, `"ShapedNoise"`, `"Default"`
`MP2.ID3`	Set to 1 to enable writing ID3v1 tags to .MP2 files (default: 0).
`MP2.JointStereo`	Set to 0 to prevent or 1 to allow Joint Stereo-mode for .MP2 files (default: 1).
`MP2.AuxEnergy`	Set to 1 to store energy level as frame aux-data for MPEG layer II (default: 0).
`MP3.ID3`	Set to 1 to enable writing ID3v1 tags to .MP3 files (default: 0).
`MP3.ID3v2`	Set to 1 to enable writing ID3v2 tags to .MP3 files (default: 0).
`MP3.APE`	Set to 1 to enable writing APE tags to .MP3 files (default: 0).
`MP3.JointStereo`	Set to 0 to prevent or 1 to allow Joint Stereo-mode .MP3 files (default: 1).
`AAC.ID3`	Set to 1 to enable writing ID3v1 tags to .AAC files (default: 0).
`AAC.ID3v2`	Set to 1 to enable writing ID3v2 tags to .AAC files (default: 0).
`AAC.APE`	Set to 1 to enable writing APE tags to .AAC files (default: 0).
`MPEG.FindLength`	Set to 0 to prevent parsing MPEG audio streams in order to determine the total length. This makes MP2/.MP3/.AAC-files open faster but the length is unknown before the conversion completes. (default: 1)
`BWF.LevelChunk`	Set to 1 to enable writing EBU "levl" (peak level info) BWF files (default: 0).
`BWF.LevelChunkBlockSize`	Set EBU 'levl' graph block size in samples (default: 256).
`BWF.LevelChunkAbsPeak`	Set to 1 to store only abs. peak value for EBU 'levl' graph (default: 0).
`VOX.SampleRate`	Set default sample rate for input .VOX-files.
`G726.BigEndian`	Set 1 for big-endian, or 0 for little-endian packaging of G.726 code words.

The `"License"` property is very important – it is used to tell the AwCOM library your license keys which in turn 'unlocks' various modules of the library. As value you pass it one or more "key=value" strings (separated by blank space if more than one, and values containing space delimited in "quotes like this"). It does not matter if you pass all your license keys in on call or one at a time in multiple calls. But the first key you pass must be "name" with the value set to your company name as given to us. Here's a typical example:

```
awManager.SetProperty("License", "name=My Company Name");
awManager.SetProperty("License", "wav=LKS34DF723DF4JASDFADF...");
awManager.SetProperty("License", "aiff=097FA5DSG23LKDD2S3AS...");
et c!
```

```
HRESULT EnumFileReaders( [in, out] long *plId,
                         [out] BSTR *pstrExt, [out] BSTR *pstrDesc,
                         [out, retval] BOOL *pbRet );
```

Enumerate available "FileReader" filters.

Note: The number of available file reader filters depends on what license keys you have set.

Note: A file reader filter can be thought of as an 'input file format handler'. You normally don't need to refer to a file reader filter by name – when you add a file reader filter (using `IAwGraph.AddFitler`*) you simply specify 'FileReader'. The library will then figure out which of its available (internal) file reader filters is appropriate to handle the specified input file and use that.*

Input:
      `*plId`   File reader filter enumeration id, start with 0 then increase the *returned value* by 1 until no there's no more file reader filters to enumerate.

Output:
      `*plId`   Enumeration id for the returned file reader filter (may differ from the input value). Note: *Never 'hard code' id values!* Ids will change if later license more file formats!

      `*pstrExt`   File name extension normally used by the file format handled by this file reader filter. Pass `NULL` if you don't care.

      `*pstrDesc`   The file format name. This is a descriptive text that can be presented to the user. It is also used to uniquely identify a file format reader – i.e. use this name and not the enumeration id if you need to identify a certain format. Pass `NULL` if you don't care.

      `*pbRet`   `TRUE` if a file reader filter was enumerated, `FALSE` if no more are available. *Note:* C++ programmes can pass `NULL` if they want and instead use the `HRESULT` code – which will be `awOk` if one was enumerated and `awOkFalse` if not.

```
HRESULT EnumFileWriters( [in, out] long *plId,
                         [out] BSTR *pstrExt, [out] BSTR *pstrDesc,
                         [out, retval] BOOL *pbRet );
```

Enumerate available "FileWriter" filters.

Note: The number of available file writer filters depends on what license keys you have set.

Note: A file writer filter can be thought of as an 'output file format handler'. When you add a file writer filter (using `IAwGraph.AddFitler`*) you specify 'FileWriter' which acts as a pseudonym for all available file reader filters. If do not also explicitly specify the file format when adding a file writer, then the library will try to find a file writer by matching the file extension. But the recommended method is supply it with a parameter "fileformat=<name>" or "id=<id>" where the name or the id is found using this enumeration function.*

Input:

*`plId`	File writer enumeration id, start with 0 then increase the *returned value* by 1 until no there's no more file writer filters to enumerate.

Output:

*`plId`	Enumeration id for the returned file writer filter (may differ from the input value). Note: *Never 'hard code' id values!* Ids will change if later license more file formats!
*`pstrExt`	File name extension normally used by the file format handled by this file writer filter. Pass NULL if you don't care.
*`pstrDesc`	The file format name. This is a descriptive text that can be presented to the user. It is also used to uniquely identify a file format writer – i.e. use this name and not the enumeration id if you need to identify a certain format. Pass NULL if you don't care.
*`pbRet`	TRUE if a file writer filter was enumerated, FALSE if no more are available. *Note:* C++ programmes can pass NULL if they want and instead use the HRESULT code – which will be awOk if one was enumerated and awOkFalse if not.

```
HRESULT EnumFileWriterFormats(
        [in] long lId, [in, out] long *plIndex,
        [out] BSTR *pstrName,
        [out] short *psMinChannels, [out] short *psMaxChannels,
        [out, retval] BOOL *pbRet );
```

For a given file writer filter, enumerate the available 'data formats'.

Input:

lId	File writer filter id as returned by `IAwManager.EnumFileWriters`.
*plId	Data format enumeration id, start with 0 then increase the *returned value* by 1 until no there's no more data formats for this file writer.

Output:

*plId	Enumeration id value for the returned data format (may differ from the input value). Don't use this value for anything except this enumeration function.
*pstrName	The data format name. This is a descriptive text that can be presented to the user. It is also used to uniquely identify the data format (within the context of the specified file writer). Pass NULL if you don't care.
*psMinChannels	Minumum no of audio channels for this data format. Pass NULL if you don't care.
*psMaxChannels	Maximum no of audio channels for this data format. Pass NULL if you don't care.
*pbRet	TRUE if a file writer data format was enumerated, FALSE if no more are available. *Note:* C++ programmes can pass NULL if they want and instead use the HRESULT code – which will be awOk if one was enumerated and awOkFalse if not.

```
HRESULT EnumAudioInputs( [in] long lId,
            [out] BSTR *pstrDesc, [out, retval] BOOL *pbRet );
```

Enumerate installed audio input devices.

Note: **Available only with a license for the AudioInput module.**

Input:

    `lId`   Enumeration id, start with 0 then increase by one until it returns `FALSE`.

Output:

    `*pstrDesc`   Name of the audio input device as given by DirectSound.

    `*pbRet`   `TRUE` if a device was enumerated, `FALSE` if no more devices.

                      Note: C++ programmes can pass `NULL` if they want and use the `HRESULT` code instead – which will be `awOk` if one was enumerated and `awOkFalse` if not.

```
HRESULT EnumAudioOutputs( [in] long lId,
            [out] BSTR *pstrDesc, [out, retval] BOOL *pbRet );
```

Enumerate installed audio output devices.

Note: **Available only with a license for the AudioOutput module.**

Input:
    `lId`   Enumeration id, start with 0 then increase by one until it returns `FALSE`.

Output:
    `*pstrDesc`   Name of the audio output device as given by DirectSound.
    `*pbRet`   `TRUE` if a device was enumerated, `FALSE` if no more devices.
                            Note: C++ programmes can pass `NULL` if they want and use the `HRESULT` code
                            instead – which will be `awOk` if one was enumerated and `awOkFalse` if not.

```
HRESULT EnumPlugIns( [in] long lId,
            [out] BSTR *pstrName, [out, retval] BOOL *pbRet );
```

Enumerate installed (compatible) VST and DX plug-in filters.

*Note: **Available only with a license for the PlugIns module.***

Input:
    `lId`   Enumeration id, start with 0 then increase by one until it returns `FALSE`.
Output:
    `*pstrName`   Name of a VST or DX plug-in filter.
    `*pbRet`   `TRUE` if a plug-in was enumerated, `FALSE` if no more plug-ins. *Note:* C++ programmes can pass `NULL` if they want and use the `HRESULT` code instead – which will be `awOk` if a filter was enumerated and `awOkFalse` if not.

```
HRESULT CreateGraph( [out, retval] IAwGraph **ppGraph  );
```

Create an empty AwGraph object and return its IAwGraph interface.

Output:

      `*ppGraph`   Pointer to the `IAwGraph` interface of the new object.

```
HRESULT CreateEasyGraph(
                [in] BSTR strInFile, [in] BSTR strOutFile,
                [in, defaultvalue(-1)] long lIdWriter,
                [in, defaultvalue("")] BSTR strWriterFormat,
                [in, defaultvalue(-1)] long lOutSampleRate,
                [in, defaultvalue(-1)] long lOutChannels,
                [out, retval] IAwGraph **ppGraph  );
```

Create an `AwGraph` *object and set it up as a completed graph for performing a conversion from one file format to another (with optional sample rate and channel format conversion), then return it's* `IAwGraph` *interface.*
Note: The only thing you then need to do with this graph is to call `IAwGraph.Execute`*!*

Input:

`strInFile`	Name and path of input file.
`strOutFile`	Name and path of output file.
`lIdWriter`	Id of file writer format as returned by `IAwManager.EnumFileWriters`. Use `-1` to auto-detect the format from the file name extension.
`strWriterFormat`	Name of a file writer data format as returned by `IAwManager.EnumFileWriterFormats`. Use `NULL` or "" to use the default data format for the file format.
`lOutSampleRate`	Sample rate of output file, use `-1` to retain the sample rate of the input file.
`lOutChannel`	Number of audio channels in output file, use `-1` to retain the sample channel format as the input file.

Output:

`*ppGraph`	Pointer to the `IAwGraph` interface of the new object.

```
HRESULT DescribeError(   [in] long lErrorCode,
                         [out, retval] BSTR *pstrDesc  );
```

This function will return an English language text description of an error code.

Input:
    `lErrorCode`   Error code value returned by an AwCOM 3 function.

Output:
    `pstrDesc`   Receives a text description of the error, it will be of the semantic type:
           E.g. `"Could not open input file"`, or
           e.g. `"Invalid or corrupted file"`, or
           e.g. `"Unsupported compression scheme encountered"`.
           If not an error code, it is set to `"No error"`.
           If not generated by AwCOM it is set to `"COM Error "`.

The IAwGraph interface

The `IAwGraph` is the main interface to an `AwGraph` object representing a filter graph.
All functions return an HRESULT value.
The actual syntax used by a particular language is somewhat language dependent. E.g. parameters marked with `[retval]` will be returned as function results in languages that hides the COM HRESULT values.

As all COM interfaces, it inherits from `IUnknown` and thus have `QueryInterface, AddRef` and `Release` methods – it is assumed that you are familiar with these – they are not documented here (note that some languages, e.g. VisualBasic, will handle reference counting for you and will hide these methods).

```
HRESULT AddFilter(  [in] BSTR strFilterName,
                    [in, defaultvalue("")] BSTR strParamList );
```

This method adds a new filter to the filter graph. The new filter is added at the end of the filter chain (i.e the output of the previous filter added, if any, will be connected to the input of the newly added filter). The first filter added must always be a 'source filter'. The last added must be a 'sink filter'. Not until after a sink filter has been added can you call 'IAwGraph.Execute'.

Input:

`strFilterName`   Filter name.

`strParamList`   Optional filter parameters. This is specified as a list of zero, one, or more 'key=value' pairs separated by space. Values containing space must be enclosed in citation marks, e.g. `file="c:\temp\out.mp2" dataformat="112 kbit/s"`. Note that some parameters may be required while others are optional to specify!

On the following pages are detailed the various filters available.

Source filters

Filter name: "`FileReader`"

Description: An audio file format reader – the library will select an appropriate filter from the available file reader filters. Note that the actual file formats that can be read depends on what license keys you have set (see further `IAwManager.SetPropery("License", …)`).

Please note that AwCOM will immediately access the specified file when adding this filter – it must not be locked by any other app or the call to add the filter will fail.

Parameter key:	*Description of parameter value:*
`file`	Input file – full name and path *(required)*.
`fileformat`	Input file reader name *(optional)*. This is the descriptive name of the desired file reader as given by `IAwManager.EnumFileReaders`. Specifying this overrides the automatic file format detection and forces the use of a specific reader filter. NB, put the format name inside quotation marks (e.g. fileformat ="my format").
`id`	Like `fileformat` but using reader enumeration id instead of name *(deprecated)*.

Filter name: "`WaveSynth`"

Description: This source filter generates a simple synthesized waveform and is intended for testing purposes.

Parameter key:	*Description of parameter value:*
`type`	Waveform type *(optional, default:* "`sine`"*)*. Allowed values are: "`zero`", "`sine`", "`square`", "`triangle`", "`sawtooth`", "`spike`".
`rate`	Waveform rate in Hz, i.e. number of cycles per second *(optional, default:* `440`*)*.
`samplerate`	Output sample rate in Hz *(optional, default:* `44100`*)*.
`length`	Output length in number of samples *(optional, default:* `44100`*)*. Specify 0 to keep running forever.

Filter name: "`AudioInput`"

Description: Audio recording source. *(NB: Available only with a license for the AudioInput module).*

After you have completed the graph, to start recording, simply call `IAwGraph.Execute()` which will return as soon as recording has been started. Recording will continue until you either free the graph, or you call `IAwGraph.SetProperty("AudioInput.Record=0")`. You can query if it is recording at any time by calling `IAwGraph.GetProperty("AudioInput.IsRecording")`, which will return "`1`" while recording, and "`0`" otherwise.

Parameter key:	*Description of parameter value:*
`device`	Audio input device name *(default: use system default device)*. Use `IAwManager.EnumAudioInputs` to find device names.
`samplerate`	Input sample rate in Hz *(optional, default:* `44100`*)*.
`id`	Like `device` but using the audio-in enumeration id instead of name *(deprecated)*.

Sink filters

Filter name: "`FileWriter`"

Description: Audio file format writer meta filter'. It will select among the available file writer filters and add an appropriate one. Note that the actual file formats that it can be written depends on what license keys you have set (see further `IAwManager.SetPropery("License", …)`).

Please note that AwCOM *may* immediately try to create the specified file when adding this filter – it must not be locked by any other app or the call to add the filter will fail.

Parameter key:	*Description of parameter value:*
`file`	Output file – full name and path *(required)*.
`fileformat`	Output file writer name *(optional but recommended)*. This is the descriptive name of the desired file format writer as given by `IAwManager.EnumFileWriters`. Specifying it overrides the automatic file writer assignment by file extension. NB, put the file format name inside quotation marks (e.g. fileformat ="Vorbis Ogg stream").
`dataformat`	The output data format *(optional, the default value depends on file format)*. Many file writer filters (specified by the `fileformat` parameter) can save the actual audio data in several different data storage formats (e.g. "`PCM 16-bit`" or "`mu-Law`"). Use `IAwManager.EnumFileWriterFormats` to find out which data formats that a particular file writer supports!
`id`	Like `fileformat` but using writer enumeration id instead of name *(deprecated)*.

Filter name: "`NullSink`"

Description: This sink filter simply swallows the input and is intended for testing purposes only.

Parameter key:	*Description of parameter value:*

Filter name: "`AudioOutput`"

Description: Audio playback sink. *(NB: Available only with a license for the AudioOutput module).*

After you have completed the graph and want to start playback, simply call `IAwGraph.Execute()` which will return as soon as playback has been started. Playback will continue either until the end, or until you free the graph, or call `IAwGraph.SetProperty("AudioOutput.Play=0")`.The latter will pause the playback. You can un-pause it by calling `IAwGraph.SetProperty("AudioOutput.Play=1")`. Use `IAwGraph.GetProperty("AudioOutput.IsPlaying")` to query if it is currently playing - it will return "`1`" when playing, and "`0`" otherwise. You can also change the playback position by calling `IAwGraph.SetProperty("AudioOutput.PlayPos=<asamplenumber>")`, or query the current position by calling `IAwGraph.GetProperty("AudioOutput.PlayPos")`.

Parameter key:	*Description of parameter value:*
`device`	Audio output device name *(default: use system default device)*. Use `IAwManager.EnumAudioOutputs` to find device names.
`id`	Like `device` but using the audio-out enumeration id instead of name *(deprecated)*.

Morph filters

Filter name: "`ChannelBroker`"

Description: Converts the input to a desired number of output channels.

Parameter key:	*Description of parameter value:*
`outchannels`	The desired number of output channels *(required)*.

Filter name: "`Trimmer`"

Description: Trims the incoming audio data and only forwards the portion inside the trim range [start … stop).

Parameter key:	*Description of parameter value:*
`start`	All incoming samples before this sample number will be removed *(default: 0)*.
`stop`	This and all subsequent samples will be removed *(default: keep all until end of data)*.
`fade`	Fade in volume during these number of samples after trim start *(default: 0)*.

Filter name: "`PlugIn`"

Description: Encapsulates VST and DX plug-ins. (NB: Available only with a license for the PlugIns module).

Parameter key:	*Description of parameter value:*
name	Name of the VST or DX plug-in filter *(required)*. Find the name using `IAwManager.EnumPlugIns.`
load	A full path to a file from which to load filter settings *(optional)*.
save	A full path to a file to which to save the filter settings *(optional)*.
edit	Show an edit dialog where the user can change settings *(optional, default:* "`no`"*)*.

Filter name: "`Normalizer`"

Description: Normalizes audio data. (NB: Available only with license for Normalizer module).
This is a 2-pass procedure – data is buffered in a temporary file on disc between the two passes. In addition to either modifying the audio data or outputting gain adjustment meta data, it will also output peak level meta data.

Parameter key:	*Description of parameter value:*
type	Normalization algorithm *(optional, default:* "`peak`"*)*. Supported values:
	`scale` Scale the volume by the target level.
	`peak` Normalize the peak value to the target level.
	`rmspower` Normalize root of mean of square of signal power to target level
	`rmsamp` Normalize root of mean of square of signal amplitude to –"–.
	`replaygain` Normalize using the ReplayGain algorithm (relative to ref.level).
	`replaygainx` Normalize using enhanced ReplayGain (w. ISO 226:2003 filter).
	`leqrlb` Normalize using –Leq(RLB) as per ITU BS.1770.
	`leqr2lb` Normalize using –Leq(R2LB) as per ITU BS.1770.
	`ebur128` Normalize using EBU R128 integrated loudness measure.
action	Set to "`modifyaudio`" to modify the volume of the actual audio samples.
	Set to "`metadata`" to output gain adjustment meta data (audio is untouched).
	(optional, default: "metadata" for replaygains, "modifyaudio" for the others)
level	The target level *(optional)*. It's interpretation depends on the "type" param: for "scale" it is in dB *(def.: "0")* – for replaygain's it is dBSPL of the ref.-level *(def.: "89")* – for leq's it is the leq-target level *(def.: "-20")* – for "ebur138" it is in LU *(def.: "0", note: 0 LU = -23 LUFS)* – for all other types it is in dBFS *(def.: "0.0")*.
value	Same as level, but given as a linear amplitude instead of in dB (value = $10^{(level / 20)}$).
maxpeaklevel	Enforces that the peak value level is not raised above `maxpeak` in dBFS. Set to "`0`" to prevent the level to be raised above the clipping point *(opt., def.: "" = off)*.
maxpeakvalue	Same as mapeaklevel but given as a linear amplitude instead of in dbFS *(opt.)*.
truepeak	Find true-peak level (w. 16x upsampling; output meta-data). *(optional, def.: "no")*
fixdc	Remove any DC offset. Values: "`yes`" or "`no`" *(optional, default: "no")*.
limiter.value	Use limiter to prevent clipping. Set to "" to clip to disable the limiter and simply clip to the -1..1 range. Set to a value between "`0`".."`1`" to enable the limiter and set the linear level above which the limiter kicks in. *(optional, default "")*
limiter.level	Sane as limiter.value, except that it's given in dBFS instead of linear level.

Note: If you wish to calculate the ReplayGain track-gain adjustment value without actually doing a conversion (i.e. without running the whole graph), then first add your file reader, then add any processing filters that you wish to work before the gain calculation, then add a normalizer filter with a type of rgmeta, rgmodify, rgxmeta, or rgxmodify, then you can ask the graph for the following properties:

Normalizer.GainAdjustment	The calculated gain adjustment in dB. NB, this is only available for these normalization types: replaygain, replaygainx, ebur128
Normalizer.InPeakValue	The peak linear amplitude of data coming into the normalizer filter.
Normalizer.InPeakLevel	The peak level in dB of incoming data coming into the normalizer
Normalizer.OutPeakValue	The peak linear amplitude of outgoing data leaving the normalizer.
Normalizer.OutPeakLevel	The peak level in dB of outgoing data leaving the normalizer filter.

NB, when calling GetProperty(), don't forget to check the return code to see if there was any error when reading from and analysing the source file.

Filter name: "Resampler"

Description: Resamples (i.e. changes the sample rate) of the audio data.

Parameter key:	*Description of parameter value:*
samplerate	The new sample rate in Hz *(required)*.
algorithm	Resampling algorithm *(optional, default:* "auto"*)*. Supported algorithms: "nearest", "linear", "cubic", "fir8", "fir12", "fir16", "fir20", "fir24", "auto".

The FIR algorithms have increasing complexity with increasing number:

- "fir8" - should be perfectly adequate for 8-bit PCM output but not otherwise recommended unless you are in a hurry. It provides > 8-bit S/N ratio near the Nyquist-frequency (and better than that at lower frequencies).

- "fir12" - is a good compromise between calculation speed and audio quality, suitable for quick conversion of 16-bit PCM output. It provides > 12-bit S/N ratio near the Nyquist-frequency (and better than that at lower frequencies).

- "fir16" - should have more than enough accuracy for perfectly handling 16-bit PCM output precision. It provides > 16-bit S/N ratio near the Nyquist-frequency (and better than that at lower frequencies).

- "fir20" - should have more than enough accuracy for perfectly handling 20-bit PCM output precision and it will work very well for 24-bit PCM output too. It provides > 20-bit S/N ratio near the Nyquist-frequency (and better than that at lower frequencies).

- "fir24" - is for when you need the best possible accuracy for 24-bit PCM output. It is by computational necessity very slow (CPU-hungry). It provides > 24-bit S/N ratio near the Nyquist-frequency (and better than that at lower frequencies).

- "auto" – selects between "fir16", "fir20" and "fir24" depending on the data format of the input file.

Filter name: "SilenceRemoval"

Description: Removes silent sections from the audio. (NB: Available only with a license for SilenceRemoval).

Parameter key:	*Description of parameter value:*
level	Thresh-hold level *(optional, default:*"0.05"*)*. A value between 0 and 1 – sections of audio data where the sample data value amplitudes are lower then this value (for at least length samples in a row) will be 'cut'.
length	Minimum number of samples in a row that must be below the thresh-hold level before an audio section is cut away *(optional, default:*"1000"*)*.
leadonly	Option to remove only leading silence (i.e. only silence at the very start of the recording will be cut). Values: "yes" or "no". *(optional, default* "no"*)*.

```
HRESULT EnumFilters(      [in] long lId, [out] BSTR *pstrName,
                          [out, retval] BOOL *pbRet      );
```

Enumerate the filters in the filter graph.

Input:

    `lId`   Filter index in the graph, use 0 for the first filter, then increase by to get the next and so on until the function returns `FALSE`.

Output:

    `*pstrName`   Returned filter name, pass `NULL` as input if you are not interested.

    `*pbRet`   `TRUE` if a filter was enumerated, `FALSE` if there's no more filters.

                      Note: C++ programmes can pass `NULL` if they want and use the `HRESULT` code instead – which will be `awOk` if a filter was enumerated and `awOkFalse` if not.

```
HRESULT Execute( void );
```

Execute the graph, i.e. perform the conversion.
Note: The call is synchronous, i.e. it will not return until the conversion is completed (or failed).

```
HRESULT GetProperty(       [in] BSTR strName,
                           [out, retval] BSTR *pstrValue );
```
Get an informational property.

Input: `strName`   Name of property to get.
Output: `*pstrValue`   Property value (And empty string if property not available).

Supported properties for retrieval:

Property name:	*Property value description:*
`In.SampleRate`	Sample rate of input file.
`In.Channels`	Number of channels of input file.
`In.Length`	Number of samples in input file. NB: "Unknown" if length is unknown!
`In.Duration`	Length in seconds (i.e. In.Length/In.SampleRate).
`In.FileFormat`	File format of input file.
`In.DataFormat`	Data format of input file.
`Out.SampleRate`	Sample rate of output file.
`Out.Channels`	Number of channels of output file.
`Out.Length`	Number of samples in output file.
`Out.Duration`	Length in seconds (i.e. Out.Length/Out.SampleRate).
`Out.FileFormat`	File format of output file.
`Out.DataFormat`	Data format of output file.
`Progress`	The current progress of an ongoing conversion in percent, i.e. 0...100.
`Meta.Title`	'Name' or 'title' given to the recording
`Meta.SubTitle`	Sub-title, or sub-heading given to the recording
`Meta.Artist`	Artist name.
`Meta.Composer`	Composer name.
`Meta.Performer`	Performer name.
`Meta.Conductor`	Conductor name.
`Meta.Publisher`	Publisher name.
`Meta.Engineer`	Recording engineer name.
`Meta.Album`	Name of product (CD, DVD et c) containing the recording.
`Meta.TrackNumber`	Track number of recording.
`Meta.Genre`	Genre/category as "(ID3v1 number)Name".
`Meta.Date`	When it was released: YYYY or up to YYYY-MM-DD, HH-MM-SS
`Meta.Comment`	Any comments for the recording
`Meta.Copyright`	Any copyright information.
`Meta.URL`	Universal resource locator reference.
`Meta.LoopBeg`	Loop start point.
`Meta.LoopEnd`	Loop end point.
`Meta.LoopType`	Loop type ("fwd", "bid", or "rel").
`Meta.RootKey`	MIDI root key number (0..127).
`Meta.FineTune`	Fine tuning in cents (-50..+50).
`Meta.GainAdjustment`	Playback gain adjustment in dB.
`Meta.AlbumGainAdjustment`	Album mode playback gain adjustment in dB.
`Meta.PeakValue`	Peak sample value (in normalized linear scale).
`Meta.Reference`	Recording reference/catalogue number or similar.
`Meta.CuePoints`	Cue points list, format: sampleno="text", nextsampleno="next text", ...
`Meta.TimeCode`	Time for broadcast (in seconds since midnight).

Note that these properties can not be retrieved until at least a source filter has been added to the graph.
The Out* properties can not be retrieved until a FileWriter filter has been added to the graph.
Note: Additional properties are available when an Normalizer, AudioInput, or AudioOutput filter is present in the graph – for details please see the documentation for these specific filters.

```
HRESULT SetProperty(     [in] BSTR strName,
                              [in] BSTR pstrValue );
```

Set an informational property.

Input:

    `strName`   Name of property to set.

Output:

    `pstrValue`   Property value.

Supported properties that you can set:

Property name:	*Property value description:*
`In.SampleRate`	Override sample rate of input file.
`ClearGraph`	Set to 1 to release all filters in the graph & free file handles.
`Meta.*`	Please refer to IAwGraph::GetProperty

Note that these properties can not be set until at least a source filter has been added to the graph.

Note: Additional properties are available when an Normalizer, AudioInput, or AudioOutput filter is present in the graph – for details please see the documentation for these specific filters.

Note: For use with BWF files, SetProperty and GetPropery supports the following name aliases:

Property name alias:	*Maps to property:*
`BWF.Description`	`Meta.Comment`
`BWF.Originator`	`Meta.Publisher`
`BWF.OriginatorReference`	`Meta.Reference`
`BWF.OriginationDateAndTime`	`Meta.Date`
`BWF.TimeReference`	`Meta.TimeCode`