awC++ v4.4

The Awave Streaming Audio Converter C++ library

Reference manual

Table of Contents:

awC++ release 4.4

Introduction:

The Awave Streaming Audio Converter C++ library, or awC++ for short, is a C++ software package that provides functions for converting between different audio waveform file formats. It supports a customizable and easily extendable list of file formats. The package has its origins in the code developed for FMJ-Software's well-known 'Awave' series of audio software.

The file License.pdf contains the details of your licensing agreement for this library.

/ *Markus* / *FMJ-Software*, https://www.fmjsoft.com/

Organization

Code organization overview

The library consists of a number of C++ classes. These implement a filter graph architecture, plus miscellaneous management functions. The graph connects a file reader filter in one end, to a file writer filter in the other end, with various conversion and processing filters in between. It then streams the data in packets through the graph during a file conversion.

The files compromising the library are organized as follows:

directory:	*contents:*
Bin	Binary builds
Doc	Documentation
Src	Base class source files
Src\Codecs	Codec modules
Src\Formats	File format modules
Src\Samples	Sample application

The ReadMe.txt file in the main directory contains a complete file listing for the particular library configuration (file format modules) supplied for each particular licensee.

The only header file you should normally need to include is *awC++.h*.

All global names used starts with the letters *aw* in order to avoid name conflicts with your own code. Same with the names of all source code files.

Each file format is implemented in a file located in the *Src\Formats* directory.

Other important files are:

awAccessories.cpp	Implements additional filter classes
awFormatTable.cpp	Manages an internal table of file format handlers.
awManager.cpp	Implements a generic access to, and manipulation of, file format filters.
awModules.cpp	Implements core filter classes.
awQuantizer.cpp	Sample bit depth quantization support.
awResampling.cpp	Resample (sample rate conversion) support.

Compilers and preprocessor options

A sample project for Visual Studio 2022 is supplied with the library (awc_vc2022.sln). If you make your own project for Visual Studio, be sure to define the __VISUALC__ pre-processor symbol globally in order to correctly configure the library.

For other platforms, you must do a little more work. First create a Makefile to suit your compiler. Use __ANSICC__ preprocessor if your platform is not Win32. Your compiler must support the "#pragma pack(1)" directive to pack data structures on byte boundaries only. You may also have to edit the "awPlatform.h" file There you must define either __BIGENDIAN__ for big endian processors (Sparc, Motorola…) or __LITTLEENDIAN__ for little endian processors (Intel, Alpha…). You must also make sure that the BYTE, SBYTE, WORD, etc. declarations map to the correct data types (this is both compiler and target platform specific).

The following optional pre-processor directives can be set in awPlatform.h:

AW_USEDOUBLEPRECISION	Use 64-bit double instead of 32-bit float as internal sample format.
AW_USE32BITSIZE	Define if you don't need to handle files larger than 2 GB.
AW_USENODIRECTSTREAMCOPY	Disables "direct stream copy" for a slightly smaller binary.
AW_USENOIMAGEDATA	Disables "image meta data" support (embedded cover images).
AW_USENOSAMPLERDATA	Disables "sampler meta data" support (loop points, root keys etc).
AW_USENOWCHAR	Define if you must use char instead of wchar_t for text data.

The project can be built with Win32 (Intel 32-bit), x64 (Intel 64-bit) and ARM64 (Arm 64-bit) as targets.

How to use the library

You can be more or less ambitious in how you use awC++. The least ambitious way – but enough for many uses – requires only three or four lines of code. A more ambitious use involves setting up and managing the conversion filter graph on your own, as well as querying and setting various properties.

Adding the library to your project

First of all, you must provide the linker with the object code. The easiest way to do this is to simply include the precompiled AwCpp.lib that comes with the library. If you want to do any changes in the library code, or use a calling convention other than the standard-C calling convention, then you will have to recompile it using the project set up provided with the library, or include the relevant source code directly into your project. The relevant files being all the .cpp files provided excluding only those found in the Src\Samples directory.

Next, set the pre-processor include-file search path to include the Src directory, then add #include "awC++.h" in the C++ files where you want to use the library.

A few notes on how to compile and use the library under Windows:

- The library does not need, nor use, C++ exception handling, nor run-time types.
- The library can be used in multi-threaded applications because it does not make use of any writeable global symbols. But you must to take care to synchronize access to object instances.

The short-cut to doing audio file format conversions

If you just want to convert files and don't want to mess with the details, here's how you can do it using only a single very high-level function call – and if that is enough, then you can ignore the rest of this manual!

A simple example should serve to illustrate:

```
#include "awC++.h"

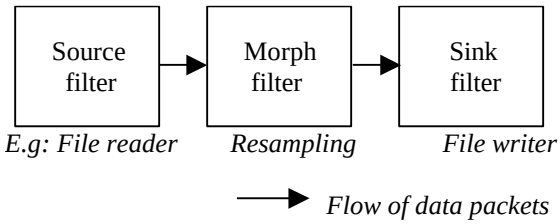
awManager aw;

AWRESULT res = aw.Convert(L"InputFile.wav", L"Output.au",
                          L"mu-Law", 1, 8000);
if (awFailed(res)) wprintf(L"Error: %s", awDescribeError(res));
```

This code will convert a WAV format input file into a Java-compatible μ-law data format mono AU file at 8000 Hz. The 1st parameter is the input file, the 2nd the output file, the 3rd the output data format (e.g. "PCM 16-bit", "PCM 8-bit", et c, use nullptr to let the program decide), the 4th is the number of output channels (1=mono, 2=stereo, 0=let the program decide), and the 5th is the desired output sample rate in Hz (0=let the program decide=same as input).

What are those filter graphs?

Filter graphs consists of modules called filters, connected in a sequential order, in order to stream small packets of data from one end of the graph to the other as illustrated in the following figure:



A source filter is any filter that has an output, a sink filter is one that takes an input and a morph filter is one that is both a source and a sink, i.e., it morphs, changes, process, monitors or just hands on the data from its input to its output.
The typical example of how this is used in awC++ is to start with a file reader filter (a subclass of source filter), connect that to a channel converter filter and a resampling filter (if needed), then finally a file writer filter (a subclass of sink filter). To convert a file, start with telling the file reader to 'read' it – which doesn't actually read in any sound data, it just reads meta data (e.g. recording length and sample rate) from the file header. After that, you can query the file reader for various kinds of information. Next, tell the file writer to write to an output file in a particular data format. It will then ask upwards in the filter chain for data packets. When the query reaches the reader, it will deliver a packet downwards in the chain until it reaches the writer, which finally writes it to disk.

Setting up and using filter graphs

The procedure for setting up and doing a file format conversion is basically the following:

- Create a file reader and read info from the source file
- Create a file writer and connect it to the file reader
- Insert any morph filters that you need (resampling, channel converter, progress indicator…)
- Tell the writer to write the output file
- Delete the graph

This section gives you a quick overview – for more details please refer to the reference section.

The `awManager` class is used to help create the read and write filters and to help manage the graph. We have already looked at the high-level `Convert` function. To create reader and writer filters you use the `CreateReader` and `CreateWriter` functions, e.g.:

```
awManager aw;
awReaderFilter* pReader;
aw.CreateReader(pReader, "MyInFile.wav");
```

You can also supply a second `AWFFID` argument to `CreateReader`. This explicitly tells the library what file format the input file is. If you do not, then the library will attempt to auto-detect it from the file. This is not always possible with certain raw data formats that neither have a file header, nor use a standardized file extension.

The `AWFFID` is a number that you get from a call to either `EnumReaders(…)` or `FindReader()`, e.g:

```
int iConfidence;
AWFFID idIn = aw.FindReader("MyInFile.wav", &iConfidence);
```

The `EnumReader` function can be used both to enumerate all file formats (see the sample code in the reference section) and for getting information about a given format, e.g:

```
const char* pszExt, * pszDesc;
aw.EnumReaders(idIn, pszExt, pszDesc);
printf("The input file is of type '%s' with %d %%
confidence\n",
       pszDesc, iConfidence);
```

There are similar `EnumWriters`, `FindWriter`, and `CreateWriter` functions for file writer filters, e.g.

```
awWriterFilter* pWriter;
aw.CreateWriter(pWriter, "MyOutFile.au", "PCM 16-bit");
```

The third parameter to the `CreateWriter` function tells it what data format should be used in the output file (most file formats supports multiple data formats). Omitting it, will let the library select this by itself. If not, then you must supply one of the formats that you enumerate by `EnumWriterDataFormats(…)`. If the writer filter doesn't support the data format that you specified then the call will fail. Note that each file format module supports its own set of data formats and you must enumerate them for each format to find out which are supported and exactly what 'strings' are used to name them.

Once you have created a reader, the next step is to read the header information from the file by calling `Read`. After that you can query it for various properties of the source file, e.g.

```
pReader->Read();
wprintf(L"Input: %d samples, %dHz, %dch, %s\n",
     pReader->Length(), pReader->SampleRate(),
     pReader->Channels(), pReader->GetTextInfo(aw_idInfoDataFormat));
```

To be able to write a file, you need to connect the reader and the writer to complete the graph. You can also create and insert any morph filters that you want in between them, e.g.:

```
pWriter->ConnectInputTo(pReader);
// Ensure mono
pReader->InsertAtOutput(new awChannelBrokerFilter(1));
// Resample to 8000 Hz
pWriter->InsertAtInput(new awResamplingFilter(8000));
```

The just described operation is so common that awManager provides a special function to do it:

```
aw.ConnectToWriter(pReader, pWriter, 1, 8000);
```

The last two parameters are optional. This doesn't just connect the filters – it also figures out if the channel format output of the reader is an acceptable input to the writer. If not it inserts a channel broker filter if necessary. Same thing goes for inserting a resampling filter.

Now you're ready to write the output file. This is simple! Just call:

```
pWriter->Write();
```

When you're done, you need to delete all the filters. Rather than explicitly call `delete` for each one of them, you can do the same more easily by calling (for any of the filters in the graph):

```
pReader->DeleteAll();
```

Error handling has been omitted until now. Most functions return an error code of type `AWRESULT`. This can be used to get more information about an error if one has occurred, e.g.:

```
AWRESULT res = FunctionCall();
if (awFailed(res)) wprintf(L"Error: %s\n",
awDescribeError(res));
```

Here's the whole process again as simple as possible:

```
#include "awC++.h"

awManager aw;
AWRESULT res;
awReaderFilter* pReader;
awWriterFilter* pWriter;

// Create reader
res = aw.CreateReader(pReader, L"MyInFile.wav");
if (awFailed(res)) CleanupAndExitWithError(res);

// Read info from input
res = pReader->Read();
if (awFailed(res)) CleanupAndExitWithError(res);

// Create writer
res = aw.CreateWriter(pWriter, L"MyOutFile.au");
if (awFailed(res)) CleanupAndExitWithError(res);

// Connect graph
res = aw.ConnecToWriter(pReader, pWriter);
if (awFailed(res)) CleanupAndExitWithError(res);

// Write output
res = pWriter->Write();
if (awFailed(res)) CleanupAndExitWithError(res);

// Clean up
pWriter->DeleteAll();
```

A more extensive command line converter sample application called AwConv is included with the library (see Src\Samples\AwConv.cpp).

Reference section

Data types and constants reference

The function result type – AWRESULT

Most functions return an `AWRESULT` value. This works just like the 'HRESULT' codes in Microsoft APIs. It is a 32-bit value that has the highest bit set if there was an error, or clear if the call succeeded. The low order bits return is a result code. Here are the currently defined codes:

Error code:	*Error description:*
`AW_S_OK`	*No error*
`AW_E_GENERAL`	*General failure*
`AW_E_INVALIDPARAM`	*Invalid function call parameter*
`AW_E_OUTOFMEMORY`	*Out of memory*
`AW_E_FILEOPEN`	*Couldn't open input file*
`AW_E_FILECREATE`	*Couldn't create output file*
`AW_E_FILEREAD`	*Couldn't read from input file*
`AW_E_FILEWRITE`	*Couldn't write to output file*
`AW_E_INVALIDOPSEQUENCE`	*Invalid function call sequence*
`AW_E_UNSUPDATAFORMAT`	*Unsupported data data format*
`AW_E_UNSUPCHANNELFORMAT`	*Unsupported channel format*
`AW_E_UNSUPFILEFORMAT`	*Unsupported file format*
`AW_E_UNSUPCOMPRESSION`	*Unsupported compression format*
`AW_E_INVALIDFILETYPE`	*Invalid or unsupported file type*
`AW_E_CORRUPTEDFILE`	*Corrupted input file*
`AW_E_EMPTYFILE`	*Empty input file*
`AW_E_INVALIDLICENSE`	*Only used by AwCOM*
`AW_E_MISSINGCOMPONENT`	*Missing external DLL, e.g. WMA codec or MP3 encoder.*
`AW_E_CANCELLED`	*Operation prematurely cancelled (by user or similar)*
`AW_E_NORIGHTS`	*No read-rights (due to DRM or similar copy-protection scheme)*

You can use `the awIsOk(code)` and `awFailed(code)` function to test if a function call succeeded of failed. Use `awDescribeError(...)` to get an English-language description string for an error code.

File format id's - AWFFID

Each file format reader and writer have its own 'id-number', of the type `AWFFID`. Not that this may change if you add a new file format module, so never had code the number! A reader and a writer for the same format may share the same number - but not necessarily! If you need to persistently store e.g. a file format selection, then store it by the reader or writers descriptive name, instead of the id, then look up the id from the name (c.f. awManager class).

Sizes - AWSIZE

This is either a 32-bit integer or a 64-bit integer used to indicate sizes, lengths, file positions, and sample positions. NB; if you want to handle files > 2GB then you must use 64-bit sizes (which is also the default).

Bit depth quantization

The internal data precision is be default 32-bit floating point. Whenever the data needs to be converted to a lower bit depth (e.g. 8-bit linear PCM), a quantizer function is called. Several different algorithms are provided for this:

`awQuantizerType::RoundNearest`

This simply rounds to the nearest sample value available for the output precision. This is the fastest and most commonly used method. *Note:* If you push in e.g. 16-bit PCM data, and don't use any processing option, and save to an equivalent 16-bit PCM data format, then you will get a bit-exact copy of the audio data. *Note:* When decreasing the bit-depth, when using any processing (e.g. resampling) that modifies the audio samples, then the quantization errors will be 'systematic', which may result in 'overtones' for strong frequency components (and thus manifest as a change of the 'timbre').

`awQuantizerType::RectWhiteNoise`

White noise (i.e. noise with a 'flat' power-spectrum) is added below the least significant bit (lsb) of the signal, which makes the quantization error random (and thus eliminates the over-tone problem). *Note:* Because the amplitude of the noise is only +/-0.5 lsb (with a rectangular 'probability distribution function' (pdf), i.e. where all values in the range is equally probable), i.e. below the output resolution of 1 lsb, you will still get a bit-exact copy when e.g. going from 16→16-bits with no processing. *Note:* The trade-off for a better subjective sound quality (no ringing of overtones) when decreasing the bit depth, is that the noise appears as very soft wide-band 'hiss' (you can easily hear this if writing to 8-bit PCM).

`awQuantizerType::TriWhiteNoise`

White noise with a rectangular pdf (i.e. where a value less probable the further from 0 it is) added to the signal. *Note:* This has theoretically been shown to be optimal type of noise to use for dithering, i.e. the subjective sound quality should be slightly better than for the previous option. *Note:* You will not get a bit-exact copy when e.g. going from 16→16-bits.

`awQuantizerType::ShapedNoise`

This is much like the previous method, but the quantization error is now fed into a feedback filter that shapes quantization noise so that it becomes louder at frequencies where the human ear is less sensitive, and softer where it is most sensitive. *Note:* The hearing thresh-hold "equal loudness contour" of the ISO 226:2003 standard (a modern day improvement to the old "F-weight curves"), is used to shape the noise, i.e. it attempts to distribute the noise frequency spectrum so that it becomes optimally inaudible. *Note:* This is professional quality noise dithering - very similar to what's used by SACD, or by audio CD's that you use bearing '20-bit' stickers.

Which method is used is controlled by the global variable `aw_eQuantizerType`. E.g.

```
aw_eQuantizerType = awQuantizerType::ShapedNoise;
```

NB: The library also has a feature called “Direct Stream Copy”. This allows data to be passed unmodified from source filter to sink filter, in its original data format (e.g. PCM 16-bit interleaved channel stereo). In this case, no format conversion and no quantization is done, which saves CPU time. Note that this is available only for a select handful of data formats, and only when writing to exactly same format as it is in the source (input file) *and* no audio processing is done (i.e. no resampling etc.). If you want to disable this feature (which makes the code size a bit smaller), then define AW_USEONLYNORMALSTREAM

Filter base classes reference

This section explains all the major filter base-classes and their usage. These classes cannot be instantiated (created with 'new') by themselves but you they provide a 'generic' way of handling filters and you will often see pointers to them used instead of a pointer to any specific 'derived' class.

The source filter base class - awSourceFilter

This filter is a base class for all other filters that has an output. Useful functions are:

- `void ConnectOutputTo(awSinkFilter* pSink);`

  Connects a sink filter to this filters output.

- `void InsertAtOutput(awMorphFilter* pMorph);`

  Inserts a morph filter 'between' this filter and the filter that is currently connected to its output.

- `awSinkFilter* Output();`

  Returns a pointer to the filter connected to this filters output, or nullptr if none.

- `void DeleteAll();`

  Delete all filters in the graph connected to this filter – saves you from having to delete them individually.

- `const char* GetTextInfoLatin1(awTextInfoId id);`
- `const BYTE* GetTextInfoUTF8(awTextInfoId id);`
- `const wchar_t* GetTextInfoUnicode(awTextInfoId id);`

Returns a string containing text meta data information.

The string is zero terminated and the format is either Latin-1 (ISO 9959-1), Unicode with UTF-8 encoding, or Unicode with UTF-16 encoding, depending on which function you call.

nullptr is returned if the queried meta data type is not available (a "" string will never be returned).

The following meta data information type 'id's are currently defined:

`FileName`	Source file name
`FileFormat`	Source file format
`DataFormat`	Source data format
`Title`	Main title or 'name' given to the recording
`SubTitle`	Sub-title of recording
`Artist`	Name of artist or performer appearing in the recording
`Composer`	Composer of music
`Performer`	Performer of music
`Conductor`	Conductor of music
`Publisher`	Publisher of music.
`Engineer`	Recording engineer's name.
`Album`	Name of product (CD, DVD, et c) containing the recording
`TrackNumber`	Track number of a song taken from a recording.
`Genre`	Song genre/category as: "(ID3v1 number)Name"
`Date`	Released date: YYYY or up to YYYY-MM-DD, HH-MM-SS
`Comment`	Any comments for the recording
`Copyright`	Any copyright information
`URL`	Universal resource locator reference
`TrackGainAdjustment`	Gain adjustment for the recording, dB
`AlbumGainAdjustment`	Gain adjustment for an entire album, dB
`ProgrammeLoudness`	EBU R 128 Programme loudness
`LoudnessRange`	EBU R 128 Loudness range
`PeakValue`	Maximum (normalized) peak sample value, -1.0 .. 1.0
`Reference`	Recording reference/catalogue number or similar
`CuePoints`	Cue points list, sampleno="text", nextsampleno="next text", ...
`TimeCode`	Time for broadcast (in seconds since midnight)
`LevelInfo`	EBU level chunk description

The sink filter base class - awSinkFilter

This filter is a base class for all other filters that has an input. Useful functions are:

- `void ConnectInputTo(awSourceFilter* pSource);`

  Connects a source filter to this filters input.

- `void InsertAtInput(awMorphFilter* pMorph);`

  Inserts a morph filter between this filters and the filter currently connected to its input.

- `awSourceFilter* Input();`

  Returns a pointer to the filter connected to this filters input, or nullptr if none.

- `void DeleteAll();`

  Delete all filters in the graph connected to this filter – saves you from having to delete them individually.

The morph filter base class - awMorphFilter

This filter is a base class for all other filters that has both an output and an input.

Inherits multiply from: `awSourceFilter` and `awSinkFilter`.

The file reader filter base class - awReaderFilter

This base class provides the common functionality for all file format readers. A pointer to a class of this type (that actually points the sub-classed implementation of a given file format) is normally obtained from (and the object is created by) the `awManager` class. When the object is created, a file is always tied to it. If you later want to read a different file, you have to create a new file reader tied to that file.

Inherits from: `awSourceFilter`.

- `AWRESULT Read();`

  Call to 'read' the file that is connected to the object. Normally this is the first thing you do after creating the reader (by calling awManager::CreateReader). This does not read the actual sound data – it reads header information such as file length, sample rate, text info et c. You should always call this function before calling any of the functions below.
  Returns `AW_S_OK` if successful, or an error code otherwise.

- `DWORD Length();`

  Returns the length of the input file in number of sound samples.

- `DWORD Channels();`

  Returns the number of channels of audio data in the input file.

- `DWORD& SampleRate();`

  Returns the sample rate in Hz (samples per second) of the input file.
  Note that it is a reference that is returned; this means that you can override the value stored in the file.

- `void SetTextInfoLatin1(awTextInfoId id, const char* pcszText);`
- `void SetTextInfoUTF8(awTextInfoId id, const BYTE* pcmszText);`
- `void SetTextInfoUnicode(awTextInfoId id, const wchar_t* pcwsz);`

  Sets or overrides a text meta-data information string for the file.

  The string is zero terminated and the format is either Latin-1 (ISO 9959-1), Unicode with UTF-8 encoding, or Unicode with UTF-16 encoding, depending on which function you call.

  See `awSourceFilter::GetTextInfo` for the definition of `awTextInfoId`.

  *Note: The text you set by this call is *not* stored in the file, it is just kept in memory and handed over to the file writer when it asks if there is any text information to write.*

- `int NumStreams();`

  Returns the number of discrete audio clips or audio streams contained in the source file (normally only 1, but some file formats support more). The reader filter only reads one such stream or clip (use the `iStreamIdx` parmeter of `awManager::CreateReader` to specify which).

The file writer filter base class - awWriterFilter

This base class provides the common functionality for all file format writers. A pointer to a class of this type (that actually points the sub-classed implementation of a given file format) is normally obtained from (and the object is created by) the `awManager` class. When the object is created, a file is always tied to it. When you have written a file and want to write a new one, you'll have to create a new writer instance.

Inherits from: `awSinkFilter`.

- `AWRESULT Write();`

  Write the output file. This will pull in all the necessary data from the source filter that it is connected to and write it to disk. Note that the object must be connected to a 'complete' filter graph chain before this, as well as any of the following functions, are called.

  Returns `AW_S_OK` if successful, or an error code otherwise.

- `DWORD Length();`

  Returns the length of the output file in number of sound samples.

- `DWORD Channels();`

  Returns the number of channels of audio data in the output file.

- `DWORD SampleRate();`

  Returns the sample rate in Hz (samples per second) of the output file.

- `DWORD MinChannels();`

  Returns the minimum number of input audio channels required by the filter.

Filter classes reference

This section explains the non-base-class filters and their usage – what they do, how to create them, and what parameters you can supply when creating them.

The file reader and writer filters

Each file format implements its own file format readers and/or writers. The implementations are found in the files in the Src/Formats directory. E.g. awf_au.cpp for the Sun Audio format. You use the awManager class to enumerate and create the file reader and file writer filters – for reference see e.g. `awManager::CreateReader` and `awManager::CreateWriter`.

The sample rate converter filter - awResamplingFilter

This filter is used to resample the sound data, i.e. to change the number of samples and the sample rate without changing the playback 'pitch' and 'time'. As an example, if you halve the sample rate, then you have to halve the number of samples in order for it to play back during the same length of time. This can be done with more or less advanced algorithms.

- `awResamplingFilter(DWORD dwNewSampleRate,`
  `awResamplingType eType = awResamplingType::eFIR16);`

  The constructor takes the following parameters:

`dwNewRate`	*Desired sample rate.*
`eType`	*Specifies the algorithm that should be used:*
`Nearest`	*Use (1-tap) nearest neighbor algorithm.*
`Linear`	*Use (2-tap) linear interpolation.*
`Cubic`	*Use (4-tap) cubic interpolating filter (default).*
`FIR8`	*Use finite impulse response filter, > 8 bits S/N.*
`FIR12`	*Use finite impulse response filter, > 12 bits S/N.*
`FIR16`	*Use finite impulse response filter, > 16 bits S/N.*
`FIR20`	*Use finite impulse response filter, > 20 bits S/N.*
`FIR24`	*Use finite impulse response filter, > 24 bits S/N.*

A short explanation of the resampling algorithms:

- *Nearest neighbor:* Does not perform any interpolation of sample values, just grabs the sample from the 'nearest' time position sample in input data.
- *Linear interpolation:* Performs linear interpolation between the two nearest of samples.
- *Cubic interpolation:* Performs cubic polynomial fitting to the four nearest of samples.
- *FIR filters:* The 'Finite Impulse Response' filters give the best spectral accuracy and the lowest noise floor of the algorithms available here, but are also the most computationally expensive. The higher the signal to noise ratio, the slower to compute.

The channel converter filter - awChannelBrokerFilter

This filter converts the number of channels of the input to a specified number of channels in the output.

- `awChannelBrokerFilter(DWORD nOutChannels);`

  The constructor takes the following parameter:

  `nOutChannels` *The desired number of output channels:*

- `void AssignChannel(DWORD nOutChannel, DWORD nInChannel);`

  Routs `nInsChannel` *to* `nOutChannel.`

- `void MixChannels(DWORD nOutChannel,`
  `DWORD nInChannel1, DWORD nInChannel2);`

  Routs a mix of `nInChannel1` *and* `nInChannel2` *to* `nOutChannel.`

- `void SilenceChannel(DWORD nOutChannel);`

  Specify that `nOutChannel` *shall be silent.*

The default action is as follows.

Mono is converted to Stereo by duplicating the mono channel.
Stereo is converter to Mono by taking the average of the channels.
3+ input channels with 6, 7, or 8 output channels are converted to standard 5.1/6.1/7.1 format if possible.
Otherwise channels > 2 in the output are taken from corresponding input if existing or else set to 0.
Channels > 2 in the input are ignored if not available in the output.

Override it by using the class methods to route or silence channels.

The trim filter - awTrimFilter

This filter throws away or 'trims' all wavesamples outsides the trim range: [awsBeg-dwFade, awsEnd+ dwFade).

- `awTrimFilter(AWSIZE awsBeg, AWSIZE awsEnd = 0, DWORD dwFade = 0);`

  The constructor takes the following parameters:

`awsBeg`	*First sample in the trim range, 0 = don't cut any at the beginning.*
`awsEnd`	*First sample after the trim range, 0 = don't cut any at the end.*
`dwFade`	*Number of samples to fade in/out before/after awsBeg/awsEnd.*

The memory source filter - awMemorySourceFilter

This source filter reads data in several formats from a piece of memory provided by your application.

- `awMemorySourceFilter(void* pData, DWORD dwLength, DWORD nChannels, DWORD dwSampleRate, const char *pcszDataFormat);`

  The constructor takes the following parameters that specify the source memory buffer:

pData	*Points to the sound data in memory. The dimension of pData should be: 'fFormat_cast(pData)[dwLength][iChannels]*
dwLength	*Number of sound samples (per channel) pointed to by pData.*
nChannels	*Number of audio channels. The channels are interleaved in the data.*
dwSampleRate	*Sample rate of the sound data in Hz.*
pcszDataFormat	*Data type of the source sound data. Valid values are:* *"PCM 8-bit" — Signed 8-bit integers (bytes)* *"PCM 16-bit" — Signed 16-bit integers (words)* *"PCM 32-bit" — Signed 32-bit integers (dwords)* *"Float 32-bit" — Normalized 32-bit floats (–1.0..1.0)* *"Float 64-bit" — Normalized 64-bit floats (–1.0..1.0)*

The memory sink filter - awMemorySinkFilter

- `AWRESULT SetBuffer(void* pBuffer, DWORD dwBufferLength, DWORD nBufferChs, const AWCHAR* pcszDataFormat);`

  This call sets the data buffer to write to. Note that the output buffer need not necessarily be the same length or number of channels as the input data. The data will be truncated, or zero padded as necessary.
  Also note that multi-channel audio is stored in 'interleaved' format.

  Input:

`pBuffer`	*Pointer to the data buffer to receive samples.*
`dwBufferLength`	*Length, in samples, of *pBuffer (= no bytes / (channels * bytes/sample).*
`nBufferChs`	*Number of (interleaved) audio channels to store in the buffer.*
pcszDataFormat	*Data type of the buffer sound data. Valid values are:* `"PCM 8-bit"` *Signed 8-bit integers (bytes)* `"PCM 16-bit"` *Signed 16-bit integers (words)* `"PCM 32-bit"` *Signed 32-bit integers (dwords)* `"Float 32-bit"` *Normalized 32-bit floats (–1.0..1.0)* `"Float 64-bit"` *Normalized 64-bit floats (–1.0..1.0)*

  `Output:`

`<return>`	`AW_S_OK` *if successful, or an error code otherwise.* `AW_E_UNSUPDATAFORMAT` *if unsupported buffer data format.*

The progress indicator filter - awProgressIndicatorFilter

This filter is useful as a base class for constructing your own 'progress indicator' or similar by overloading the virtual `DisplayProgress` member. It can also be used to provide a 'cancel operation' function.

- `virtual void DisplayProgress(int iPercentage);`

  This function does nothing as is, but you can easily derive your own class from this class and override it! It takes this single parameter:

`iPercentage`	*A value between 0 and 100 describing how large a percentage of the file conversion has been completed so far.*

- `void Cancel();`

  This function cancels, or stops the filter running, as soon as possible. Note that it will most likely not have stopped yet when this function returns. It has stopped when the Write() call to the filter chains sink returns with the error message AW_E_CANCELLED.

Here's some sample code:

```
class OurProgressDisplay : public awProgressIndicatorFilter
{
      void DisplayProgress(int iPercentage) override;
};

void OurProgressDisplay::DisplayProgress(int iPercentage)
{
      wprintf(L"%d%%\n", iPercentage);
}
```

The generator source filter - awGeneratorFilter

This filter synthesizes simple waveforms. Could be useful for testing and reference purposes.

- ```
  awGeneratorFilter(awGeneratorType type, float fHz = 1000.0f,
                    DWORD dwSampleRate = 44100,
                    DWORD dwLength = AW_LENGTH_UNKNOWN);
  ```

  The constructor takes the following parameters:

eType	*Type of synthesized waveform. Available options are:*
	Zero — *Silent wave*
	Sine — *Sine wave*
	Square — *Square wave*
	Triangle — *Triangle wave*
	Sawtooth — *Sawtooth wave*
	Spike — *Spike/pulse wave*
fHz	*Frequency of waveform, oscillations per second.*
dwSampleRate	*Sample rate of output.*
dwLenght	*Number of wavesamples to output, default = until chain sink stops.*

- ```
  void SetFrequency(float fHz);
  ```

  This function changes the generators frequency.
  It takes this single parameter:

fHz	*Frequency of waveform, oscillations per second.*

The null sink filter - awNullSinkFilter

This filter just feeds off the connected input filter chains and throws away the wavesamples. Could be useful for testing purposes, or e.g. as the 'end' that drives a filter chain containing morph filters that e.g. collects statistics, or determined the input level, of the source data.

- `AWRESULT Write();`

  Starts the filter chain. Note that the object must be connected to a 'complete' filter graph chain before this function is called.

  Returns `AW_S_OK` if successful, or an error code otherwise.

Filter management reference

Each file format module has an initialization function whose job it is to keep a register of reader and writer filters in a global table (the initialization functions are listed in a table in the file *Src/awFormatTable.cpp*). The `awManager` class then uses this global table of filters to provide you with a generic way of enumerating and creating file format filters - without needing any a priori knowledge of what file formats are available. It also provides some useful filter related management functions.
Each file format reader and writer get its own 'id-number', declared as the type `AWFFID`. A reader and a writer of the same format may share the same number - but not necessarily! So never use numbers for 'comparison', just as parameters for the `awManager` functions.

The filter manager - awManager

- ```
  AWRESULT Convert(
     const AWCHAR* pcszInFileName, const AWCHAR* pcszOutFileName,
     const AWCHAR* pcszDataFormat = nullptr, DWORD
  nOutChannels=0,
     DWORD dwResample = 0, AWFFID idIn = AW_AUTODETECTID,
     AWFFID idOut = AW_AUTODETECTID);
  ```

  This is the definite function to use if you just want a 'high-level' way to convert from one file format to another and don't want to be bothered with any filter graph management details.

  Input:

`pcszInFileName`	*Points to the name and path of the input file.*
`pcszOutFileName`	*Points to the name and path of the output file.*
pcszDatabFormat	*A string specifying the output data format. You should use `EnumWriters()` to find the id of the writer that you want to use, then `EnumWriterDataFormats`() to find out what data formats it supports.* *Alternately specify nullptr to get the writers default data format.*
`nOutChannels`	*The number of audio channels that you want in the output file.* *Use `EnumWriterDataFormats()` to find out the minimum and maximum number of audio channels that a given format supports.* *Alternately specify 0 to use the same number of audio channels as the input file (if possible, else it'll use the nearest possible channel count).*
`dwResample`	*Resample the sound to this sample rate, 0 = keep original sample rate.*
`idIn`	*File format id from `EnumReaders()`.* *Alternately use `AW_AUTODETECTID` to auto-detect the input format by looking at the input file.*
`idOut`	*File format id from `EnumWriters()`.* *Alternately use `AW_AUTODETECTID` to detect from the output file name extension.*

  Output:

`<return>`	*Returns `AW_S_OK` if the operation was successful, else an error code.*
  ```

- `bool EnumReaders(AWFFID& id, const AWCHAR*& pcszExt, const AWCHAR*& pcszDesc);`

  Enumerates the available file format reader filters in alphabetical order.
  This is useful e.g. when constructing the format list for an 'Open File' box.

  Input:

`&id`	*Enumeration id, Set 0 for the first file format.*

  Output:

`&id`	*Id of the returned file format. Add 1 and call again to get the next one.*
`&pcszExt`	*Will be set to point to the 'standard' file type extension will be stored. The dot (.) is not included. 'Standard' here means the one most commonly used if the format is found under more than one extension. It can be more than 3 characters; if you need '8.3' compatibility you may want to truncate it.*
`&pcszDesc`	*Will be set to point to a description of the format type.*
`<return>`	*Returns `true` if a format was enumerated.* *Returns `false` if there are no more formats.*

  Sample code that prints a list of all file formats that can be read:

```
awManager aw;
const AWCHAR* pcszExt, * pcszDesc;

for (AWFFID id = 0; aw.EnumReaders(id, pcszExt, pcszDesc);
id++)
        wprintf(L".%s\t%s\n", pcszExt, pcszDesc);
```

- `AWFFID FindReader(const AWCHAR* pcszFileName, int* piConfidence = nullptr );`

  Determines the appropriate file format reader filter for an existing file pcszFileName by looking both at the extension and for 'markers' inside the file. If the file does not exist, then only the extension is used.

  Input:

`pcszFileName`	*Points to a file name or just a file extension...*

  Output:

`*piConfidence`	*A measure in the range [0, 100] of how 'certain' the file type detection was. The value 0 means 'unknown file type', and 100 means as good as absolutely certain. Pass in nullptr if you don't want to know.*
`<return>`	*A file format id if successful or `AW_INVALIDID` if the file could not be opened or if no matching reader was found.*

- `AWRESULT CreateReader(awReaderFilter*& pReader,`
  `const AWCHAR* pcszFileName,`
  `AWFFID id = AW_AUTODETECTID, int iStreamIdx = 0);`

  Opens a file and returns a reader filter for it.

  Input:

`pcszFileName`	*Points to the name and path of the input file.*
`id`	*File format id from `EnumReaders()`. Alternately use `AW_AUTODETECTID` to auto-detect the format from the file name extension.*
`iStreamIdx`	*A few file formats supports storing more than one discrete audio clip or audio stream in the same file. This parameter is a zero-based index to specify which stream or clip to read from such a file. To find out how many streams there are, create a reader for the first stream using a stream index of 0 (default), then call the `NumStreams()` of the returned reader filter.*

  Output:

`&pReader`	*Receives a pointer to an awReaderFilter if succesfull, else `nullptr`.*
`<return>`	*Returns `AW_S_OK` if the operation was successful, else error code.*

- `bool EnumWriters(AWFFID& id,`
  `const AWCHAR*& pcszExt, const AWCHAR*& pcszDesc);`

  Enumerates the available file format writer filters in alphabetical order.
  This is e.g. useful when constructing the format list for a 'Save As' box.

  Input:

&id	*Enumeration id, Set 0 for the first file format.*

  Output:

&id	*Id of the returned file format. Add 1 and call again to get the next one.*
&pcszExt	*Will be set to point to the 'standard' file type extension will be stored. The dot (.) is not included. 'Standard' here means the one most commonly used if the format is found under more than one extension. It can be more than 3 characters; if you need '8.3' compatibility you may want to truncate it.*
&pcszDesc	*Will be set to point to a description of the format type.*
`<return>`	*Returns `true` if a format was enumerated. Returns `false` if there are no more formats.*

- `bool EnumWriterDataFormats(AWFFID id, int& n,`
  `const AWCHAR*& pcszName,`
  `DWORD* pnMaxChannels = nullptr,`
  `DWORD* pnMinChannels = nullptr);`

  Enumerates the available data formats for the specified file format writer filter.

  Input:

id	*File format writer id from EnumWriters()..*
&n	*Enumeration id, Set 0 for the first data format.*

  Output:

&n	*Returned enumeration Id. Add 1 and call again to get the next one.*
&pcszName	*Will be set to point to the name of the data format.*
**pnMaxChannels*	*If not nullptr, will be to the maximum number of input audio channels supported by the data format. This is typically 1, 2, or 1000.*
**pnMaxChannels*	*If not nullptr, will be to the minimum number of input audio channels required by the data format. This is typically 1 or 2.*
<return>	*Returns `true` if a data format was enumerated.* *Returns `false` if there are no more data formats.*

- `AWFFID FindWriter(const AWCHAR* pcszFileName);`

  Determines the appropriate file format writer filter based on the extension.

  Input:

`pcszFileName`	*Points to a file name or just an extension.*

  Output:

<return>	*A file format id if successful or `AW_INVALIDID` if no matching reader was found.*

- `bool WriterSupportsDataFormat(AWFFID id, const AWCHAR* pcszName);`

  Chacks if the indicated writer filter supports the given data sub-format.

  Input:

id	*File format writer id from EnumWriters().*
pcszName	*Name of the data format to check if it is supported.*

  Output:

<return>	*Returns `true` if the data format is supported* *Returns `false` otherwise.*

- `AWRESULT CreateWriter(awWriterFilter*& pWriter,`
  `const AWCHAR* pcszFileName,`
  `const AWCHAR* pcszDataFormat = nullptr,`
  `AWFFID id = AW_AUTODETECTID);`

  Creates a writer filter and associated output file.

  Input:

`pcszFileName`	*Points to the name and path of the output file.*
pcszDataFormat	*A string specifying the output data format. You should use* `EnumWriters()` *to find the id of the writer that you want to use, then* `EnumWriterDataFormats`*() to find out what data formats it supports. Alternately specify nullptr to get the writers default format.*
`id`	*File format id from* `EnumWriters`*, use* `AW_AUTODETECTID` *to detect from extension.*

  Output:

`&pWriter`	*Receives a pointer to* `awWriterFilter` *if successful, else nullptr*
`<return>`	*Returns* `AW_S_OK` *if the operation was successful, else error code.*

- `AWRESULT ConnectToWriter(awSourceFilter* pSource,`
  `awWriterFilter* pWriter,`
  `DWORD nOutChannels = 0, DWORD dwSampleRate = 0);`

  Connects any source filter to a writer filter, inserting a channel converter filter and/or a resampling filter in between if necessary to satisfy the requirements of the writer.

  Input:

`pSource`	*Pointer to an* `awSourceFilter` *derived class*
`pWriter`	*Pointer to an* `awWriterFilter`*.*
`nOutChannels`	*The number of audio channels that you want in the output file.* *Use* `EnumWriterDataFormats()` *to find out the minimum and maximum number of audio channels that a given format supports. Alternately specify 0 to use the same number of audio channels as the input file (if possible, else it'll use the nearest possible channel count). If required, an awChannelBrokerFilter is inserted in the filter chain.*
`dwSampleRate`	*Desired sample rate of output, 0 = leave as is* *If required, an awResamplingFilter is inserted in the filter chain.*

  Output:

`<return>`	*Returns* `AW_S_OK` *if the operation was successful, else error code.*

Miscellaneous functions reference

Miscellaneous functions used for error handling and other common tasks.

- `inline bool awIsOk(AWRESULT code);`

  This tests the high bit of an error code to see if it is clear.

  Input:

`code`	*Error code value returned by an awC++ function.*

  Output:

`<return>`	`true` *if it is no error,* `false` *if it is an error.*

- `inline bool awFailed(AWRESULT code);`

  This tests the high bit of an error code to see if it is set.

  Input:

`code`	*Error code value returned by an awC++ function.*

  Output:

`<return>`	`true` *if it is an error,* `false` *if it is no error.*

- `const AWCHAR* awDescribeError(AWRESULT code);`

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

  Input:

`code`	*Error code value returned by an awC++ function.*

  Output:

`<return>`	*A pointer to a text description of the error. It will be of the semantic type: "Could not open input file", or "Invalid or corrupted file", or "Unsupported compression scheme encountered".*

- `float awSamplesToTime(AWSIZE awsPos, DWORD dwSampleRate);`

  Convert a sample index position to a time in seconds.

  Input:

awsPos	*Sample index position or length.*
dwSampleRate	*Sample rate of the audio data.*

  Output:

<return>	*Returns the corresponding time in seconds.*

- `AWSIZE awTimeToSamples(float fTime, DWORD dwSampleRate);`

  Convert a sample index position to a time in seconds.

  Input:

fTime	*Time in seconds.*
dwSampleRate	*Sample rate of the audio data.*

  Output:

<return>	*Returns the nearest corresponding sample index position or length.*

Symbol index