Adobe® Audition® Amio Software Development Kit

Introduction

Amio is the name of the file-level media IO interface used by Adobe® Audition® CS5.5. File filters or plug-ins built with previous SDK versions are no longer supported.

The heart of this SDK is the sample code, which implements an Amio plug-in supporting input and output of Monkey's Audio losslessly compressed audio files. This SDK distributes files modified from Monkey's Audio SDK version 4.06, copyright © 2000-2009 by Matthew T. Ashland. Please be aware if you use any of the Monkey's Audio source code included with the example plug-in, that use of this code is subject to the Monkey's Audio license at http://www.monkeysaudio.com/license.html.

The sample plug-in can be built using Microsoft® Visual Studio® C++ 2008, or Apple® Xcode® 3.2.5. It is possible other versions of these tools may be used to create a plug-in which is compatible with Adobe Audition CS 5.5, but only these have been tested.

For each new plug-in you create, you must generate a new identifier known as a UUID or GUID. Every plug-in must have a unique identifier, and you cannot simply use the identifier from the sample code. If you are unsure how to generate a UUID, do a web search on "uuid generator". See the use of `AmioGetAmioInfoInterface::SetPluginID` in AmioApeInterface.cpp, around line 93.

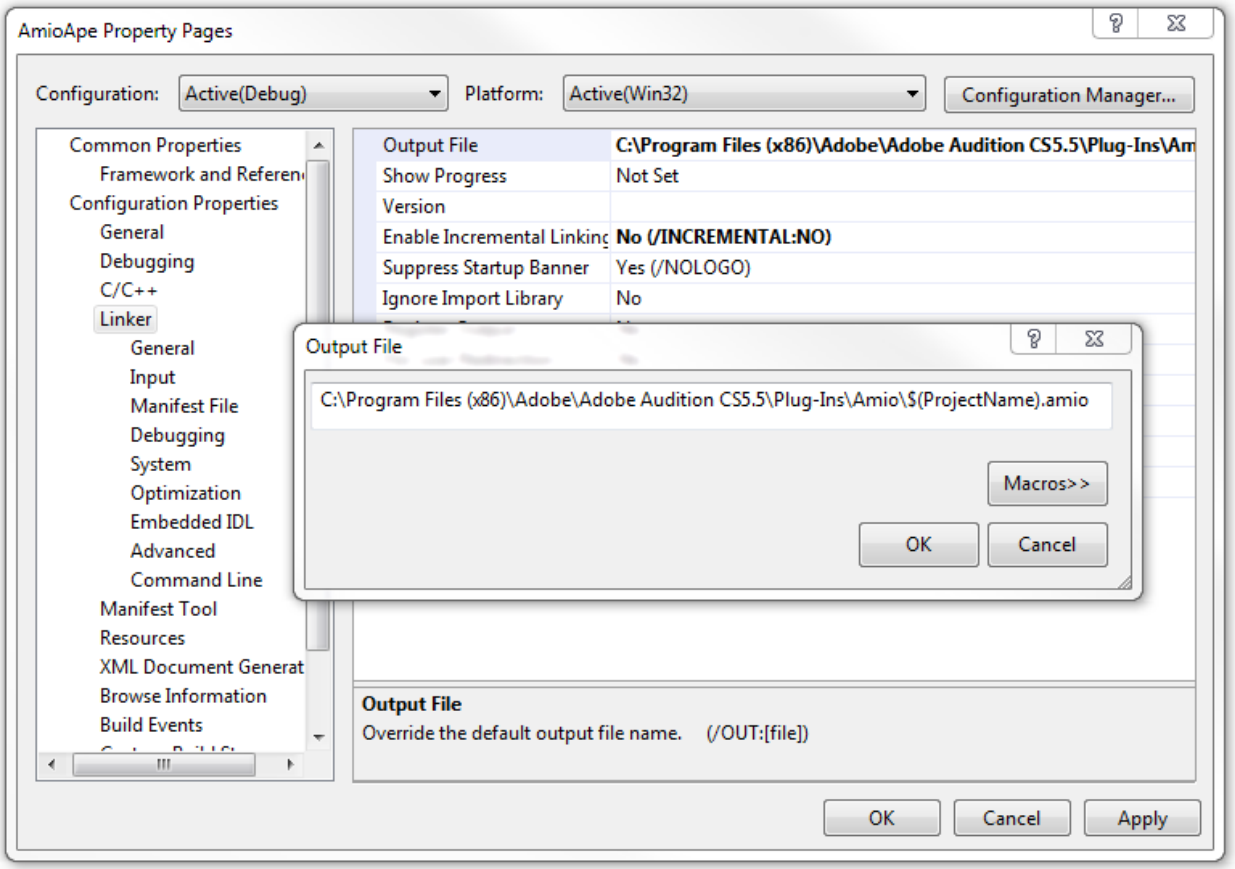
Apart from studying the sample code in general and AmioApeInterface.cpp in particular, the most important file to understand is AmioSDK.h, which defines the interfaces the plug-in uses to communicate to the host application and describes how they are used. The most complete technical documentation of the Amio SDK is found in the comments there.

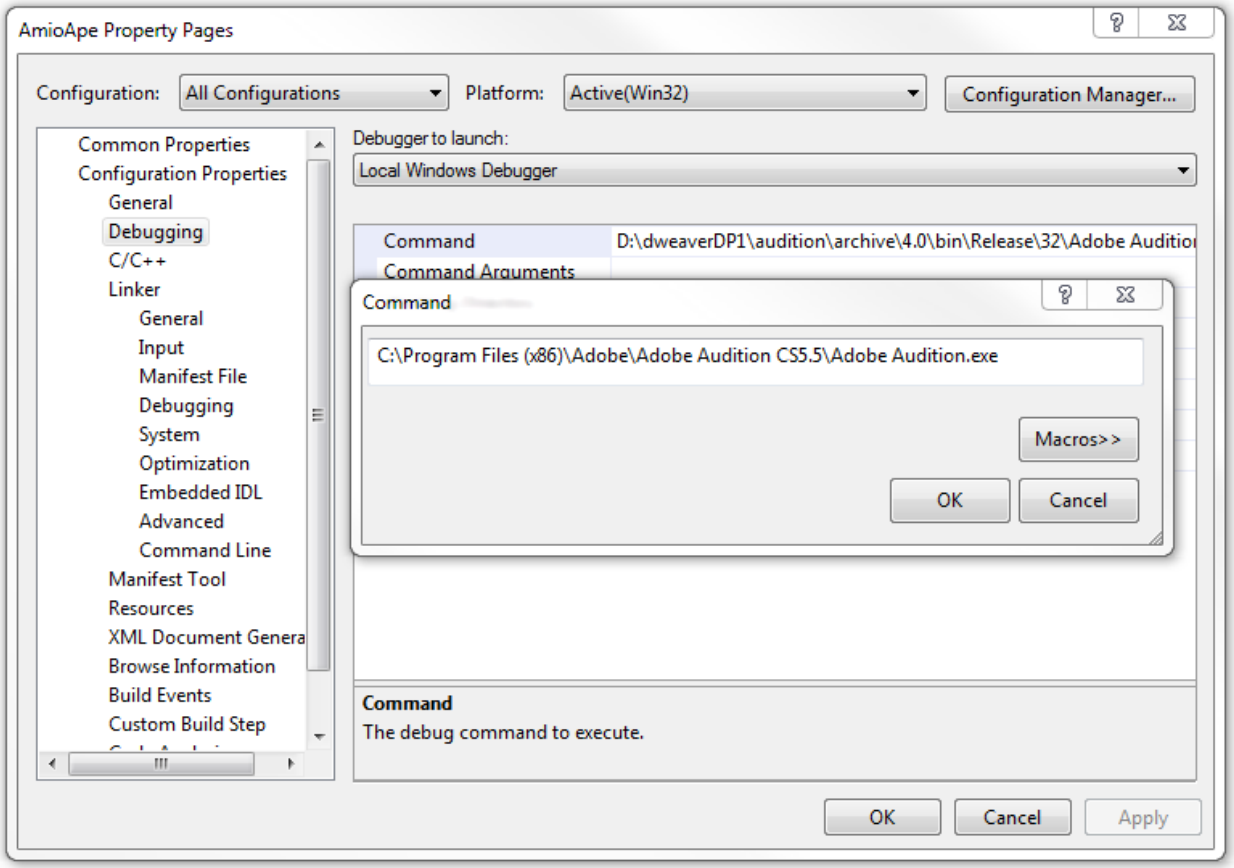
Project Set-up

Note that even debug builds of your plug-in must be compatible with the release build of Adobe Audition CS5.5. For Windows builds, you will want to use the non-debug version of the C runtime library used, `Multi-threaded (/MT)`.

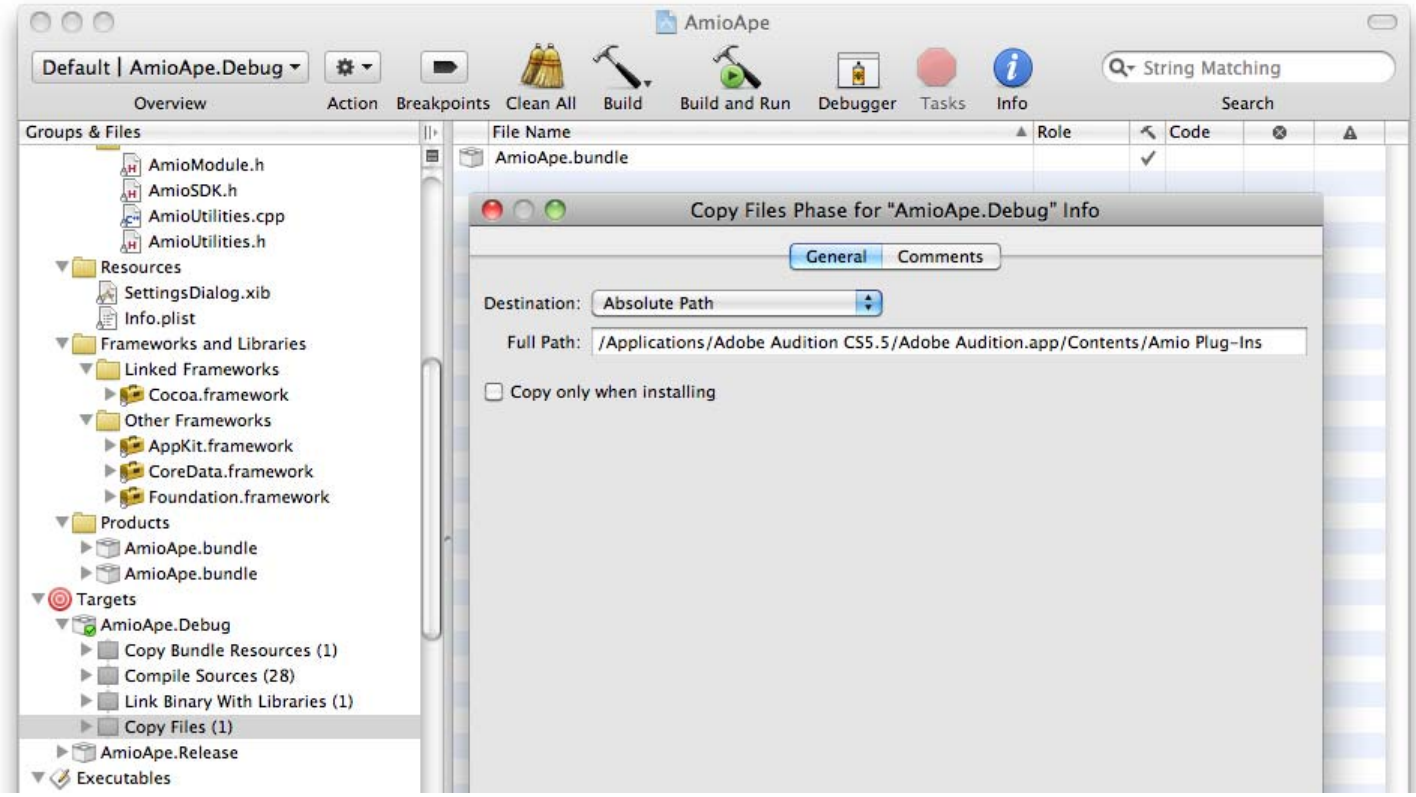
To build and run a plug-in during development, set the build target and executable directory appropriately. Note that if you want to build plug-ins on Windows directly into Audition CS5.5's default install location in the Program Files (x86) directory, you may need to run Visual Studio as administrator in order to have the proper privileges.

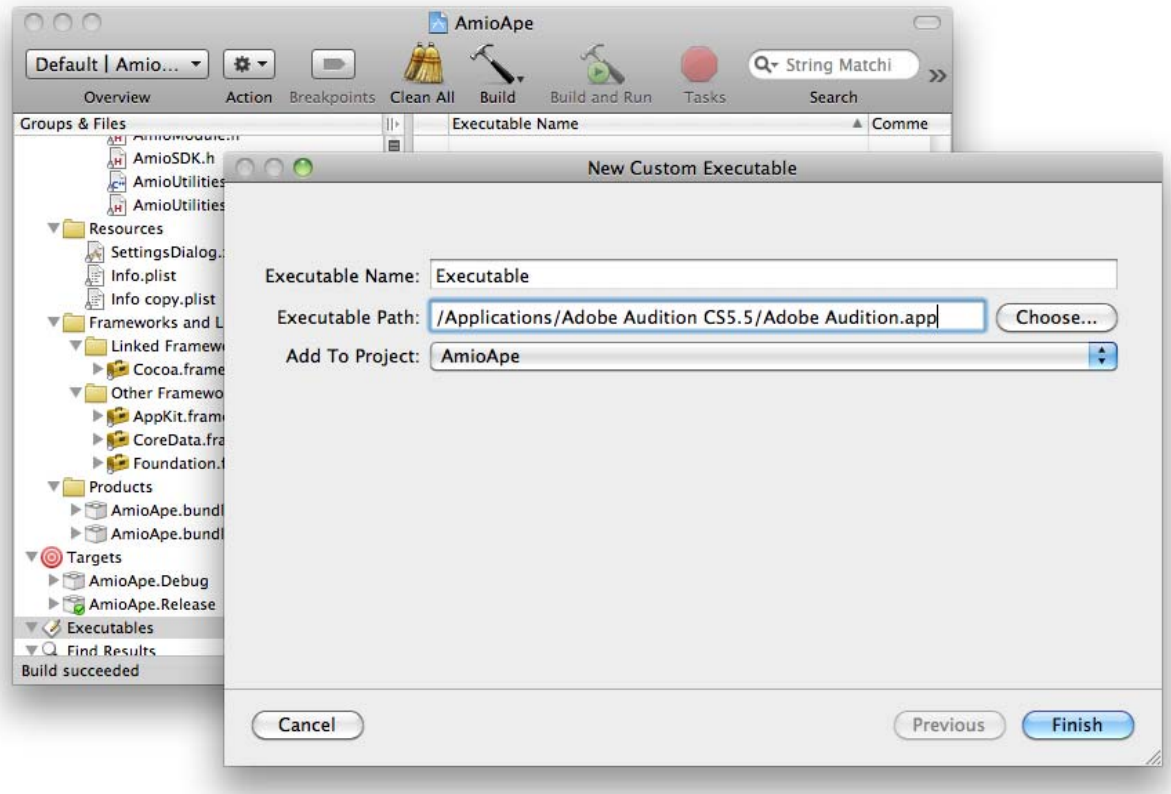
In Visual Studio, a plug-in should be given an ".amio" extension and built to the Plug-Ins\Amio directory:


AmioApe Property Pages
Configuration: Active(Debug)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Framework and Referen
Configuration Properties
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
Resources
XML Document Generat
Browse Information
Build Events
Output File
C:\Program Files (x86)\Adobe\Adobe Audition CS5.5\Plug-Ins\Am
Show Progress
Not Set
Version
Enable Incremental Linkinc
No (/INCREMENTAL:NO)
Suppress Startup Banner
Yes (/NOLOGO)
Ignore Import Library
No
Output File
C:\Program Files (x86)\Adobe\Adobe Audition CS5.5\Plug-Ins\Amio\$(ProjectName).amio
Macros>>
OK
Cancel
Output File
Override the default output file name. (/OUT:[file])
OK
Cancel
Apply


AmioApe Property Pages
Configuration: All Configurations
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
Resources
XML Document Genera
Browse Information
Build Events
Custom Build Step
Debugger to launch:
Local Windows Debugger
Command
D:\dweaverDP1\audition\archive\4.0\bin\Release\32\Adobe Auditio
Command Arguments
Command
C:\Program Files (x86)\Adobe\Adobe Audition CS5.5\Adobe Audition.exe
Macros>>
OK
Cancel
Command
The debug command to execute.
OK
Cancel
Apply

In Xcode, a plug-in should be given a ".bundle" extension and built to Contents/Amio Plug-Ins:





A Tour of the SDK Contents

- **AmioSDK** - The Amio interface definition and a few helper classes and utilities are found here.
    - AmioSDK.h - The most important file in the SDK, containing the list of Amio commands and interfaces.
    - AmioSDKTypes.h - A few types and defines, including the supported audio sample types, and channel labels (i.e. speaker placements).
    - AmioInterfaceTemplate.h - A helper template if you choose to use it (discussed below).
    - AmioUtilities - A few utilities, most notably Unicode conversions as nearly all strings passed between the app and plug-in are UTF16 strings.
    - AmioPrivateSettingsSerializer - If you wish, you can use this class to help create and parse the strings which represent settings your file type supports but are not interpreted by the app. For example, for an mp3 file, these settings would include a file's bitrate and whether CBR or VBR compression is used. The plug-in passes a setting string to the app when a file is opened. When a file is written, the plug-in receives the string from the app so that a saved file can be created using the same sub-type and settings used in the original file. If your plug-in posts a settings dialog, the settings are also passed as such a string. See `GetPrivateFormatData` and `SetPrivate-FormatData` in AmioSDK.h.
- **Common/AmioApe** - Most of the source code for the example plug-in is common between Windows and Mac, and is found here.
    - AmioApeInterface.cpp - The most important file in the example plug-in, as it contains the entire interface between the plug-in and the app. All other files merely support the methods found here.
    - AmioApePrivateSettings - A class that keeps track of the private format settings used by the plug-in, in this case merely the compression level.
    - ApeReader / ApeWriter - Classes that use the Monkey's Audio `IAPEDecompress` and `IAPECompress` interfaces to read and write audio samples and metadata.
- **Common/Monkeys Audio Modified Source** - The Monkey's Audio source code, hastily modified to support Windows and Mac.
- **Documentation** - This documentation.
- **Mac Sample/AmioApe** - The Xcode project for building the example plug-in, including a sample release build binary of the plug-in, and the Mac-specific files—mostly relating to the settings dialog.
- **Windows Sample/AmioApe** - The Visual Studio project for building the example plug-in, including a sample release build binary of the plug-in, and the Windows-specific files—mostly relating to the settings dialog.

Concepts

There is a single C interface between the app and plug-in:

```
extern "C" AMIO_EXPORT AmioResult AmioInterface(asdk::int32 inCommand,
void* inState, void* inInterface);
```

A single breakpoint here will catch all communication between the app and the plug-in. All calls are from the app to the plug-in, and because the plug-in does not signal, notify, or call back into the app, this allows for a very simple interface. Through this interface, a command is specified, a state object is passed if necessary (generally this will be an instance of an object the plug-in has created to store the state of a file being read or written), and a pointer to the interface the plug-in can use to get or set parameters related to the given command.

The SDK provides, and the sample code uses, a template class, `template<class ReadT, class WriteT> class AmioInterfaceTemplate`, which handles all the casting to specific Amio interface types, and allows you to think in terms of only the higher level Amio interfaces and your own file reader and writer classes. This template also provides default empty implementations of methods you do not need to override, and a couple helper methods such as `SetErrorString` and `AddWarning` for easily communicating error and warning text back to the app. In any case, this template is not required, and you may choose to handle all the individual low-level commands yourself. If you do use the template, your `AmioInterface` function will, as in the example code, simply pass all commands to the `EntryPoint` method of the class you derive from `AmioInterfaceTemplate`.

Each plug-in is queried using `AmioGetAmioInfoInterface`, and must provide a unique GUID to identify itself. Using this interface, the plug-in sets the human-readable name of the format it is supporting as well as a list of file extensions that can be used for reading and writing. If no extensions are set for file input, that means only writing is supported; conversely, if no extensions are set for file output, then only reading is supported.

Note: *Please be absolutely certain that you use a new, unique GUID for each plug-in you create or distribute!*

Using `AmioOpenInterface`, the app requests the plug-in to open a file and provide basic format information about it. If a file is supported and will be opened, but with caveats, warning text can be provided using `AmioErrorInterface`, and will be displayed by the app. After open, additional format information will be queried using `AmioFormatInterface`. Data will then be read from the file, generally using `AmioReadSamplesInterface`. When the app is finished with the file, it will be closed using `kAmioInterfaceCommand_Close`.

When writing a file, the app may use `kAmioInterfaceCommand_GetDefaultExportSettings` to get format parameters that the plug-in always supports for writing. Often, the app will use the `AmioGetExportSettingsInfoInterface` to pass in a format it would like to use. This format may have come from a different file type and not be valid for your plug-in. Here is where you must modify the parameters to the closest format that is supported by your plug-in. In this way, the app negotiates with the plug-in to arrive at a format that is mutually agreeable.

Note: *To give the user the best possible experience, please set all the parameters in* `AmioGetExportSettingsInfoInterface` *as accurately and completely as possible.*

After the final format and settings have been validated by the plug-in, a request is made to create a file using `AmioWriteStartInterface`. The app then passes output samples to the plug-in sequentially using `AmioWriteSamplesInterface`, and asks the plug-in to finish and close the file using `AmioWriteFinishInterface`.

Careful handling of metadata is an important courtesy to the user, and essential for any modern workflow. Metadata that is found and described by the plug-in when a file is opened is read by the app using `AmioReadMetadataInterface`. During file writing, metadata is made available to the plug-in at all times—during open, writing, and finishing—via `AmioFormatInterface::GetMetadataItem`, `GetMetadataItemInfo`, and `GetMetadataItemCount`. Thus, your plug-in can write the supplied metadata to the file at the most convenient time.

The type of each item of metadata is described by a GUID. If there is metadata that your plug-in reads that should flow through Audition CS5.5 uninterpreted and be written back out when the file is saved, assign your own GUID(s) to represent the metadata in those particular formats. If metadata is passed to your plug-in using a GUID you do not recognize, you should ignore it and not write it as if it were metadata you understand.

All the metadata the user can see and manipulate in Audition CS5.5 is contained in a metadata item with the ID `kAmioMetadataTypeID_XMP`. It is vital that your plug-in not discard or ignore this data. This metadata should be stored inside the files you support and correspondingly retrieved. The example plug-in demonstrates this by using the flag `kAmioFileFlag_XmpSupportThroughPluginOnly` and translating XMP metadata items with the tag `kAmioMetadataTypeID_XMP` to and from the RIFF style metadata the plug-in supports natively. Note that an eccentricity in the interpretation of the `kAmioFileFlag_XmpSupportThroughPluginOnly` flag means that the `kAmioFileFlag_WriteXmpMetadataBeforeSamples` flag should also be set during writing. See the example plug-in for guidance on the effective use of this flag.

If you are writing a plug-in for a file format supported by Adobe's XMP library, you may instead use `AmioFormatInterface::SetXmpHandlerId` to set the appropriate file type, causing XMP metadata reading and writing to be controlled by the app using the XMP library, outside the plug-in. But if your file format is not fully supported by the existing handlers, there is no choice but to implement XMP metadata handling in the plug-in itself.

Note: *Please do not distribute a plug-in without first verifying that XMP metadata can be correctly read from the file, viewed and modified in the Audition CS5.5 UI and written to the file.*

Functionality Not Shown in the Example Plug-in

The example plug-in demonstrates audio read and decompressed from start to finish immediately after a file is opened. This audio is cached and maintained by the app, and no further data for this file is requested from the plug-in. If a plug-in uses a format which supports instant reads and seeks, there may

be no need for this time-consuming conforming step: the plug-in can set the flags `kAmioFileFlag_ReadSamplesRaw` and `kAmioFileFlag_RealTimeSupport`. In this case, the app will request audio whenever it is needed using the `AmioReadSamplesRawInterface` interface. This interface deals not with separate channels each containing an integer number of audio samples, but rather with blocks of interleaved sample data at byte rather than sample offsets. Although this approach may be used, it is not generally recommended because of the requirements it places on the plug-in to support smooth playback in real-time.

When the exact number of audio samples cannot be known until after all audio data is processed, that is, when the sample count cannot easily be computed at the time a file is opened, the plug-in can return an estimated number of samples, set the `kAmioFileFlag_SampleTotalInexact` flag, and use `AmioReadSamplesInterface::SetEndOfFile` to effectively communicate the total number of samples to the app.

Video is supported for reading in Adobe Audition CS 5.5. When a file is opened, you can use `AmioVideoFormatInterface` to set information about video streams in the file. The app will then request video frames using the `AmioReadVideoFrameInterface`. This version of the interface supports only 32-bit ARGB video data, 8 bits per component.

The SDK supports the case in which user interaction is needed to resolve format ambiguity when opening a file. See `AmioPreOpenInterface` and `AmioRunImportSettingsDialogInterface`.

Using `AmioGetAmioInfoInterface`, the SDK exposes a way for a single plug-in to expose support for multiple types of files. That is, `AddInputFormat` or `AddOutputFormat` can be called multiple times with a different formatID. This support is not likely to work well in the current version of the app, and this approach should be avoided. Generally, a separate plug-in with a unique `SetPluginID` GUID should be used for each format you want to provide, that is, for each different collection of file extensions.

Note that this SDK distributes files modified from Monkey's Audio SDK version 4.06, copyright © 2000-2009 by Matthew T. Ashland. These modifications were made in a fairly hasty manner to allow cross-platform compilation, and in particular, as the wide character size differs between platforms, it was expedient to remove wide character support. We thank the creator of the Monkey's Audio SDK for the opportunity to use it and apologize for any errors that may have been introduced into the modified source during the process of preparing the Amio sample plug-ins.