ARA API Documentation

This pdf document describes the ARA API specification.
The ARA Library module of the ARA SDK includes an extended html version of this document which additionally covers the ARA library classes and functions.

Version 2.2.0

Introduction

Welcome

First of all, thank you for your interest in the ARA Audio Random Access API.

Celemony Software GmbH and its ARA partners put a lot of effort into establishing ARA as a mature, reliable platform. By publishing this Software Development Kit (SDK) free of any licensing fees under the Apache 2.0 License, Celemony invites more interested audio developers to join the ARA community.

Interacting with an API as complex as ARA requires careful execution by all involved parties in order to minimize bugs and incompatibilities and provide a great user experience across all the different host/plug-in combinations. Therefore, all ARA developers are requested to please follow these guidelines:

- Study the API and its accompanying documentation carefully before planning your adoption of ARA.
- If in doubt about any detail of the API, please get in touch with us via ara@celemony.com.
- Do not modify the API in any way, and stick to official API releases for your public releases.
- Test carefully with a wide range of hosts/plug-ins respectively, both in terms of the API contract and in terms of product design/user experience.
- If you experience bugs in the SDK or with a particular ARA-enabled product, notify Celemony or the product vendor respectively.
- If a bug is reported for your product, try fixing it within a reasonable time frame.
- Communication between ARA partners should be concise, friendly and respectful as laid out for in example in the Apache Code of Conduct.

Please note that ARA Audio Random Access is a registered trademark of Celemony Software GmbH. Permission to use this trademark for your product (website, boxes, etc.) may be granted free of charges after Celemony or a selected ARA development partner on behalf of Celemony has tested your implementation with respect to the above guidelines.

The purpose of this process is to leverage the valuable experience of long-time ARA developers to catch potential bugs and incompatibilities before they are released, considerably easing the maintenance of ARA-enabled products and the entire ARA platform in the long run.

About ARA

ARA Audio Random Access is an extension for established plug-in standard APIs such as VST3 and Audio Units to allow for a much-improved DAW integration of plug-ins like Celemony's Melodyne which are conceptually closer to a sample editor than to a conventional realtime audio processor. ARA does not try to replace the established plug-in standards, but rather augments them with additional features that are not present in the current versions of these standards, yet are crucial for a great user experience when using plug-ins like Melodyne.

As the name suggests, ARA's primary focus is enabling plug-ins to read audio from the host independently of the render callbacks. It needs to provide some information about the arrangement of the audio inside the host to accomplish this task.

On top of the actual audio access, ARA allows to exchange descriptions of the content of an actual audio signal, for example in terms of notes that were played when recording the signal. Advanced plug-ins like Melodyne may be capable of deriving this information by analyzing the waveform of the signal, enabling the host to build features on top of this description exchange that were simply impossible to accomplish before.

A third major feature of ARA is allowing plug-ins to be used as a time-stretching engine for the host.

The regular ARA workflow starts by the host providing random access to the audio material that it wants to be processed through an ARA capable plug-in. It specifies what parts of the audio material are to be played at what point in time. It also provides a description of the musical context such as tempo or harmonic structure in which the material will be played back.

The plug-in then performs an in-depth analysis of the material. Its graphical user interface, which typically resembles some sort of advanced ”sample editor”, will allow for adjusting the results of the analysis as the user sees fit.

Based on the (potentially edited) analysis results, the plug-in can then adjust the audio content to the musical context when rendering it. This can include both time-stretching and pitch-shifting of parts of the material as needed to create musically meaningful results.

The user can also interact further and apply whatever creative editing is desired and enabled by the plug-in - in the case of Melodyne editor that would include freely rearranging any notes contained in the audio.

Further, the results of the analysis and of the creative editing are provided to the host and it can use this information e.g. to enrich its graphical representation of the material or to feed the description to a synthesizer that duplicates the musical content of the original audio (but creates a different sound).

Opposed to established offline processing APIs such as AudioSuite or Offline Audio Units, he rendering part of ARA remains realtime capable, only the audio access and the analysis are done ”offline” on non-realtime threads.

The flexibility of the ARA API allows for a large set of features to be implemented in very different host scenarios. On the ARA home page you will find a movie that demonstrates ARA in action, illustrating some of the workflows possible.

Also, to gain a deeper understanding of plug-ins like Melodyne that allow editing on per-note-level and their potential capabilities you can check the Melodyne online manual, which also contains many more videos.

ARA has been developed by Celemony in collaboration with PreSonus.

For further information, contact Celemony: ara@celemony.com
See also https://www.celemony.com/ara.

ARA Design Overview

Technical Design Overview

The ARA API is designed to be consistent, simple-to-use and very flexible. It tries not to rely on any assumptions about the actual implementation on either side.

Various parts of the API are optional, allowing both ARA-capable hosts and plug-ins to only support those aspects of ARA that are meaningful within their particular context. Each side should strive to make as much of its functionality available to the other side as possible, but should not make any assumption about how exactly that functionality will actually be used.

A good example for this pattern is the exchange of analysis results: if asked for it, the plug-in will deliver all data it can possibly derive via its analysis capabilities, but building features like audio-to-MIDI conversion or synchronizing and quantizing live recordings based on the plug-in's analysis is entirely up to the host - it may come up with features that the plug-in's creators never thought of.

Relationship Between ARA And Established Plug-In APIs

Each implementation of ARA relies on an established plug-in standard that is used as a companion standard to the ARA API. Currently, VST3 and Audio Units are supported. The companion API defines all the usual tasks for audio plug-ins such as discovering and loading plug-in binaries, configuring and executing both realtime and offline rendering, providing a user interface for the plug-in and so on.

Lightweight extensions to the companion API then provide access to the additional ARA-specific functions for managing the ARA model, providing plug-ins with random access to the audio data, etc. These ARA interfaces are defined generically and are independent of the companion API in use, allowing developers to decouple the bulk of their ARA-related codebase from the particular companion API they are supporting.

The companion API itself will still be used strictly according to its definitions and rules, ARA does not abuse or re-define its features.

The decision to design ARA as such an extension to various established plug-in standards was made to keep the implementation efforts for both hosts and plug-ins adopting ARA as small as possible. It serves this purpose well on the technical side, but it unfortunately also blurs the lines what an ARA-enabled plug-in actually is and how it should be used.

An ARA-enabled plug-in is not simply an established realtime plug-in with some extra features, but rather a whole new breed of plug-in that has different capabilities and constraints. Both hosts and plug-ins must reflect this distinction in their implementations and conceptually separate an ARA-enabled plug-in from its non-ARA-enabled counterpart: Although they live in the same binary, each version is to be treated as a separate and incompatible plug-in.

Hosts can provide both versions next to each other and let the user choose which to insert (e.g. Logic Pro X), or can automatically prefer the ARA-enabled version if the insert point is suitable for ARA (i.e. first in chain) and fall back to the non-ARA version otherwise (e.g. Studio One). Other hosts (e.g. Cubase) implement explicit ARA and non-ARA plug-in insert points.

The Melodyne online manual provides a comprehensive description of various ARA host behaviors.

The distinction between ARA and non-ARA variant also extends to persistency: settings from the ARA version cannot be loaded by the non-ARA version and vice versa. Host documents must encode this distinction accordingly in order to be properly restorable. Documents created before the adoption of ARA must keep using the non-ARA variant.

There are several other, more subtle semantic distinctions between ARA and non-ARA plug-in usage, they will be noted throughout this document and in the ARA headers.

Language And Platform Support

For highest compatibility with existing code and to work with all established plug-in formats, the ARA interface is defined in C99, making it easy to interface with other programming languages as needed.

Because of its widespread use, extra effort was put into convenient C++11 compatibility (namespaces, constants definitions, calling conventions). The supporting library code wraps the ARA API into C++11 classes and utilities.

The ARA API itself does not depend on a certain compiler or OS version.

The ARA library code heavily relies on C++11 and is therefore set up to compile with Xcode 8 for Mac OS X 10.9 or higher and with Microsoft Visual Studio 2017 for Windows 8.1 or higher.

We've also experimented with compiling the ARA SDK on Linux (Ubuntu 18.0.4) using gcc 8.3 and clang 7.0 with the provided CMake files - see ARA_Examples/CMakeLists.txt for more details.

While largely agnostic to the underlying platform, only x86 and x86-64 compatible systems are currently tested and supported. ARM64 will be supported in a future update.

In order to provide a compiler-independent binary-compatible API, ARA needs to use types with defined sizes. It does so by specifying typedefs for all data types used in the API. These typedefs are set up depending on the compiler in use to ensure compatible data type sizes.

There are some implications of this: for one, enum types cannot be used in the API directly, because their size is implementation-dependent. Instead, for each enum a matching integer type is introduced in the API, which must always carry values from the enumeration only.

Along the same lines, we're not using `stdbool.h`, but rather define a fixed-sized ARABool for the API.

API Versioning

Most structs in the ARA API contain a member indicating their size. This is used to provide a simple versioning scheme for extending the API in a backwards-compatible way. Later revisions can append new members at the end of the struct and define default behavior that is to be assumed whenever the struct does not yet contain the new members at runtime.

Basing the versioning on the size of the struct rather than defining abstract API version numbers has the advantage of more readable code, because as a programmer you don't need to know which version of the struct contains which fields. Instead, you explicitly check for the presence of a given field by comparing the struct size, which is close to what `-respondsToSelector:` allows for in Objective-C and similar to COM's `queryInterface()`.

In addition to this backwards-compatible versioning via struct sizes, ARA also features API generations, which allow for incompatible API changes should the need arise.

For more information, see the Versioned Structs and API Generations, respectively, of the API reference.

Objects And Object References

While written in C, ARA uses an object-oriented design that can be easily bridged to and from C++ and other object oriented languages. ARA objects are identified at runtime using strongly typed, opaque references called ARA...Ref for plug-in objects and ARA...HostRef for host objects. These references are pointer-sized identifiers bound to a certain object class.

In typical implementations such as in the accompanying example code, the ARA references hold the C++ `this` pointer, but it also may contain an index into an array of structs or whatever else fits the given implementation.

Each ARA...Ref only has a meaning on the plug-in side; on the host side it is an opaque token that is simply passed along whenever calling into the plug-in. The same principle applies the other way around for ARA...HostRef. For example, objects that represent areas of audio playback are referred to on the plug-in side via a ARAPlaybackRegionRef, whereas the equivalent objects in the host are referred to via a ARAPlaybackRegionHostRef.

Actual ARA object instances are never shared between the host and the plug-in. Instead, the host uses dedicated functions to create ARA objects on the plug-in side, which are then addressed using their opaque ARA...Ref, and vice versa (ARA...HostRef). Amongst other benefits, this allows for running the plug-ins and the host in separate address spaces.

The caller of the creation function is responsible for properly managing the life time of the ARARef / ARAHostRef. and its associated ARA object by calling a matching destruction function when appropriate. Typically, this is easy to implement because both the host and the plug-in already have a life-time management for their internal model objects in place, and the additional management of the corresponding objects on the other side can be directly coupled to this existing code.

Following the `Model-View-Controller` pattern, ARA distinguishes between abstract model objects that are hardly more than ”plain-old-data” types and operational controller objects that manipulate those model objects and the relationships between them.

Defining the API boundary at controller level and representing the model objects with abstract tokens allows for maximum flexibility when interfacing ARA with a given code base: instead of having to code a set of predefined ARA model classes, existing infrastructure can be leveraged to represent the ARA model data with only minimal glue code at controller level translating between the ARA abstractions and the existing implementation.

The interface of any given controller object is expressed as a set of function pointers, which are defined according to the class of the object. This matches the concept of e.g. the `vtable` in C++, but there is no direct binary compatibility because `vtables` contain private, language and compiler implementation dependent data. Instead, ARA defines simple C structs containing the function pointers.

For an example of a simple interface, see the ARAPlaybackControllerInterface.

The ARA Document Controller

In the established plug-in standards, no model data is shared between the individual plug-in instances. This works well because the access to the audio data or the timing information is always limited to the small window of the current realtime rendering block.

With ARA however, this window is lifted and thus much more information is exchanged between the host and the plug-ins. Creating copies of this potentially memory-intense information for each plug-in instance is too expensive, so ARA avoids this by establishing the concept of a document which contains the model graph and is shared between compatible plug-in instances. ARA documents typically represent a ”song” or ”project” within the host, and hosts can support multiple documents simultaneously.

The ARA document is not managed by any of the plug-in instances, but rather by a new, separate entity called the ARA document controller in a 1:1 relationship. The document controller is the gateway to the ARA model graph - when plug-in instances are created via the companion API to perform tasks such as rendering audio or displaying model data in the UI, they will use the document controller to access the model graph.

It is possible to use an ARA document controller without any accompanying plug-in instance, e.g. when utilizing the analysis capabilities of a plug-in like Melodyne to extract tempo information from a set of audio files in a background batch process. Using a companion plug-in instance without binding it to a document controller on the other hand will disable all its ARA features because the plug-in will have no access to model data for rendering or display.

To create instances of an ARA document controller, the plug-in provides a static factory. Besides creating actual controller instances, the factory also provides meta information about the controller that is needed e.g. to find the right controller class when loading archives or to find out whether a given controller class is capable of performing a particular background analysis that the host wants to make use of.

If ARA wasn't using established plug-in APIs, then this factory would be the starting point for the contract between plug-in and host. With the companion plug-in APIs in place, ARA adds a call to retrieve the matching ARA factory for a given companion plug-in instance. By supporting this call, a plug-in also indicates towards the host that it is capable of ARA in the first place.

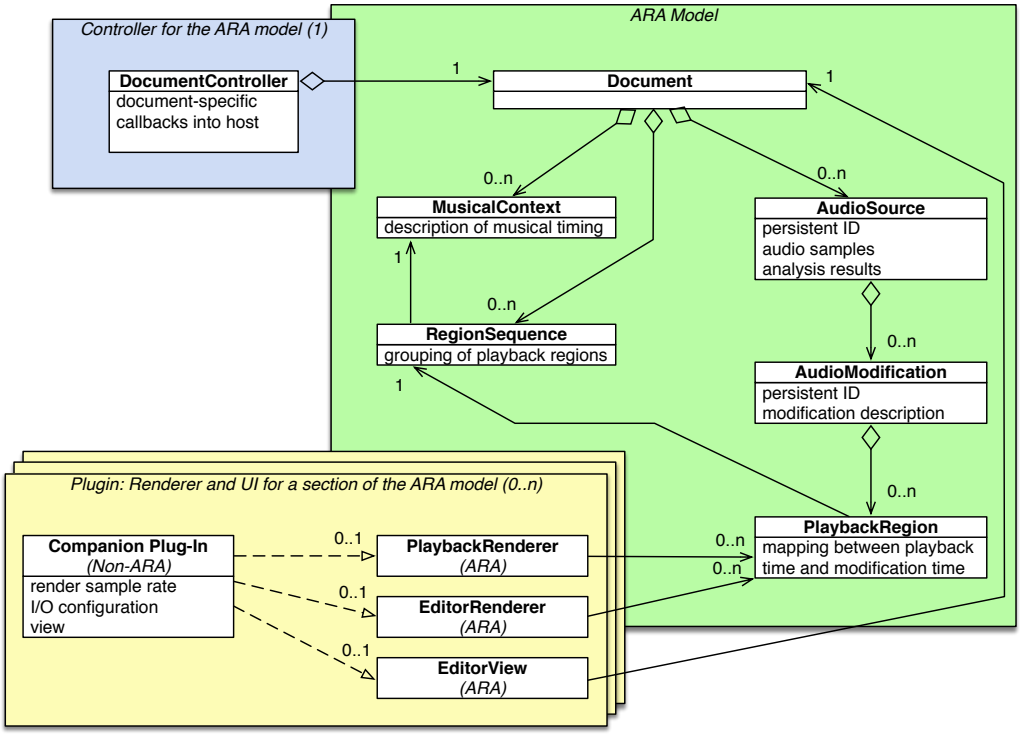
Where the companion API allows for it, ARA further defines an additional entry into the binary to access the factory independently from any plug-in instances. This may be the preferred method of discovery where available.

ARA Model Graph Overview

Once the ARADocumentController is established, the host interacts with it to publish the model graph it wants the ARA plug-in instances to process. This graph exposes the relevant subset of host model objects to the plug-in, and each of these objects is mirrored inside the plug-in. The connection between the host and the plug-in representation of each object is established via the opaque ARA object references discussed above.

The structure of the ARA model graph is entirely controlled by the host: creating and removing objects or changing connections between them is never done by the plug-in, only by the host.

The following provides an overview of the ARA model graph:



The root object of the ARA model graph is the ARADocument, managed by an

ARADocumentController in a 1:1 relationship as discussed above.

Each ARADocument can hold a set of ARAAudioSources, representing sampled audio data (such as an audio file) owned by the host. The audio source sample data should be considered immutable - while updating it is technically possible, doing so will typically invalidate all dependent data in the plug-in, described below.

The sample data will usually be analyzed by the ARA-enabled plug-in as needed to accomplish its tasks. The host also has the option to explicitly trigger any analysis it may need to implement certain features.

Note that plug-ins may feature a user interface to fine-tune analysis parameters or to manually override any incorrect analysis results. For an example of this, see `Melodyne's online manual`.

Based on the audio source analysis, the plug-in will create an internal representation of the content which is the foundation for any edits that the user can apply to the material. These subsequent edits are plug-in-specific and controlled through the plug-in UI exclusively. Their associated states are encapsulated in ARAAudioModification objects which are opaque towards the host.

Each audio modifications can be thought of as mini-arrangement of the sample data of the underlying audio source - it contains the actual creative work of the user.

Because of the tight coupling with the analysis of the underlying audio source samples, audio modification states are typically invalidated whenever these samples are updated, resulting in a loss of productive work. Modifying the audio source samples should therefore be avoided whenever possible.

Note that while an audio source covers a limited time range, an audio modification is conceptually unlimited in time - the user freely can re-arrange the audio source data, and in that process it's possible to copy&paste data before the start or behind the end of the original audio source. The host should therefore allow the user to ”resize” audio modifications arbitrarily to cover any time range.

It is possible to create more than one audio modification per audio source, so that the user can create various musical variations of the same audio source material.

A comparable pattern is already established in many advanced MIDI editors: when duplicating MIDI data inside an arrangement, the user often has the option to either create an alias that will reflect all later edits applied to the original data, or to create an individual copy of the data that will no longer be influenced by changes to the original. In terms of the ARA model, an individual copy would be represented by a new audio modification being created (”cloned”), whereas an alias would simply refer to the original modification.

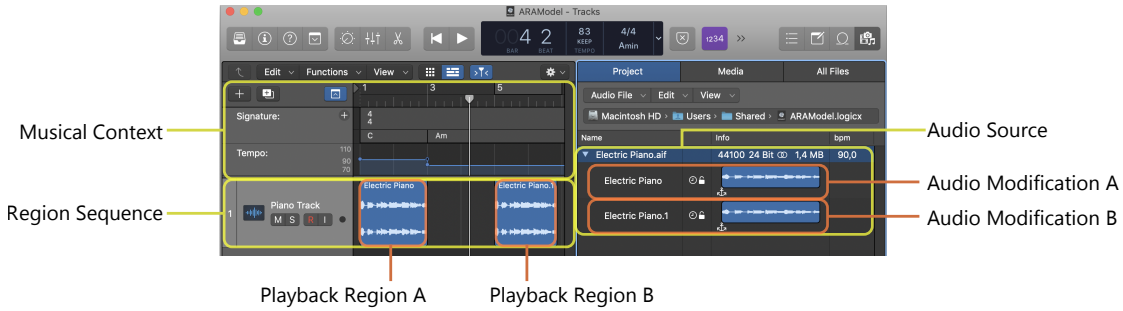
To arrange the modified audio within the song, the host will create ARAPlaybackRegions which describe segments of audio modifications that are to be rendered in a specific timing context. Playback regions can be augmented with specific transformations like time-stretching and content based fades, as detailed later on. Such a playback region usually matches objects termed ”audio region” or ”audio event” in current DAWs.

Playback regions can be grouped by the host into ARARegionSequences - typically this grouping is done according to their arrangement in the song, i.e which ”track” or ”lane” the regions are on in the song.

The term region sequence was chosen so it will not collide with the many different usages of the term ”track” across existing DAWs: channels in the mixer, lanes in the arrangement, hierarchies/folders of such lanes, or even entire songs.

In their arrangements, host usually define a musical map describing changes in tempo and scale or chord progressions. ARA encapsulates this data in an object called an ARAMusicalContext . Each region sequence (and therefore each region contained in the sequence) is associated with a musical context.

The following screenshot demonstrates how Logic Pro maps its internal model to the ARA model graph.



The ARA objects discussed have explicit and well-defined properties that are initialized by the host upon creation - these properties may be updated at any time as long as the updates occur within an edit cycle. For example, hosts may use an edit cycle to update the start and end time of a playback region or the color of a region sequence.

Both plug-in and host also have internal data structures that contain much more information about the object, such as the results of the analysis of an audio source inside the plug-in or the signatures of a song timeline inside the host. This content information may or may not be present, and the internal format for it may be completely different in each program.

To exchange this information between plug-in and host ARA establishes content readers, which are temporary objects used to iterate content information in a normalized format. They will be detailed in a later section of this document.

The structure of the model graph and the properties of the objects therein are directly controlled by the host via the document controller, and it also directly notifies the plug-in about whenever it needs to update content information due to user edits in the host.

If the content on the plug-in side is modified, the plug-in must notify the host accordingly - the host polls for these updates when it is idle.

Exchanging Content Information

A key feature of ARA is the ability to exchange information about the musical content of ARA objects. This communication can work both ways - for example the host can ask a plug-in to execute an analysis of an audio source and then query its results, but it also can provide the plug-in with information about the audio source up-front, such as the tempo map used when recording the audio.

Most importantly, the host uses this API facility to describe the content of the musical context in which the playback regions will be rendered in.

To suit the very different capabilities and needs of the various plug-ins and hosts, ARA defines a set of content types that may or may not be supported by either the plug-in or the host. Due to the sequential nature of music, each content type is encoded as a series of content events in time. Each event is represented by a particular ARA content struct associated with the content type.

The most prominent example of an content type are musical notes, which can be described via start, length, average volume and pitch just like MIDI does. ARA defines a basic note event type like so:

```
typedef struct ARAContentNote
{
    // average frequency in Hz, kARAInvalidFrequency if note has no defined pitch (percussive)
    float frequency;

    // index corresponding to MIDI note number (or kARAInvalidPitchNumber)
    ARAPitchNumber pitchNumber;
```

```
    // normalized level: 0.0f (weak) <= level <= 1.0f (strong)
    // This value is scaled according to human perception
    float volume;

    // time marking the beginning of the note (aka "note on" in MIDI), relative to
    // the start of the described object (audio source/playback region)
    ARATimePosition startPosition;

    // time marking the musical/quantization anchor of the note,
    // relative to the start of the note
    ARATimeDuration attackDuration;

    // time marking the release point of the note (aka "note off" in MIDI),
    // relative to the start of the note
    ARATimeDuration noteDuration;

    // time marking the end of the entire sound of the note (end of release phase),
    // relative to the start of the note
    ARATimeDuration signalDuration;
} ARAContentNote;
```

The amount of data used to represent content information can be very large. ARA therefore does not transfer all information about all events in a single transaction. Instead, it leverages the common iterator pattern by defining content event readers.

Code that wants to read certain content information can use a reader object bound to a given ARA model object, content type and time range. This pulling is done in small and fast transactions.

To keep the amount of transferred data small, the data structs returned by the content reader are not versioned like other ARA structs. Extending the data provided by the content readers in a new version of ARA will instead be done by simply adding new content types.

Besides providing (or not providing) the actual information, it is possible to also specify a quality for it, named ”content grade”. This is particularly useful when trying to resolve potentially conflicting information such as a plug-in detecting a tempo of 140 bpm when the host claims a tempo of 120 bpm.

For example, when making up some default value for initializing model data without any correlation to the actual music, such as assuming a default tempo of 120 bpm for a new song, this information would be accordingly qualified as ”initial”.

If the values are then determined via an analysis of the material, the grade would be upgraded to ”analyzed”. If modified by the user, it would become ”adjusted”.

There is an even higher grade of quality called ”approved” which is typically not achieved by normal editing. Instead, this would be used by content creators to explicitly mark the current state of information as ”correct” to prevent further adjustments by end-users.

It may be surprising at first that events are counted using an ARAInt32, not an ARASize data type which would extend to 64 bit on the according platforms. However this was done to emphasize that these events are entities that the user must be able to comprehend and manually edit, so 32 bits are way beyond the upper limits applicable here.

Updating The ARA Model Graph

As explained above, the structure of the ARA model graph is exclusively controlled by the host. Plug-ins will augment the ARA model graph with their internal application specific model data. Changes to this internal data may be explicitly triggered by user edits in the plug-in UI or may happen automatically, i.e when applying the results of an audio source analysis that has been executed in the background.

To prevent collisions between these various sources for changes to the graph, ARA requires that changes to the model graph must be applied serially on the main thread (either by applying them on the main thread or by blocking the main thread while applying it from a different thread).

Furthermore, edits by the host must be grouped into explicit edit cycles during which the plug-ins can suppress any pending internal edits, which is important to keep the undo history in both the host and the plug-in consistent.

Here's a code example illustrating the use of an edit cycle if the user renames the current ARA document:

```
// start the edit cycle
araDocumentController->beginEditing ();

// update the document name
ARADocumentProperties documentProperties;
documentProperties.structSize = ARA_IMPLEMENTED_STRUCT_SIZE (ARADocumentProperties, name);
documentProperties.name = "Updated Document Name";
araDocumentController->updateDocumentProperties (&documentProperties);

// end the edit cycle
araDocumentController->endEditing ();
```

Using edit cycles has a number of further benefits. First of all, it defines an explicit window of time (i.e. anytime outside the editing cycle), where both the host and the plug-in are guaranteed to be in sync, and render results thus are well-defined. This is important if a plug-in has some sort of lag between changing the model graph and actually making the results audible, e.g. because it has to update large caches - the plug-in will not return from endEditing() until these updates are finished.

Edit cycles also allow plug-ins to optimize performance when there are interdependencies between ARA objects - updates of their internal model data may be delayed until the end of the editing cycle when the new ARA graph is fully known to suppress unneccessary intermediate calculations.

Consequently, hosts should take care to group all related graph changes into a single edit cycle where possible instead of implementing multiple edit cycles which each contain only a subset of the changes..

Further, plug-in implementations may have to take special action from beginEditing() and endEditing() to ensure that any ongoing realtime or offline rendering can still access a consistent graph despite the ARA model being edited concurrently.

Note that access to audio source samples can be enabled or disabled outside of an edit cycle because this is considered a controller operation not a model edit - see the audio access section for more information.

ARA Model Persistency

When archiving an ARA model graph, three interconnected parts must be preserved: the host internal data, the resulting ARA graph structure shared between host and plug-in, and the plug-in internal data that is attached to it. As discussed above, the ARA graph structure is described using runtime object references. When restoring an ARA archive, hosts will first restore their internal model, then re-create the ARA graph. Once these ARA objects are created, the plug-in will restore their internal data.

Since the runtime references have changed since the graph was stored, the plug-in needs a way to map stored state into the newly created objects. The host therefore must define a unique persistent ID for each persistent ARA object in that object's properties. Both host and plug-in must save this ID alongside their respective data structures and use it when restoring the graph.

To preserve storage space and ease the implementation, only those ARA objects are persistent that contain information that can be modified in the plug-in. For example, the musical context is defined and

edited on the host side only. Therefore it can always be re-created from the host state when loading the document. The same goes for playback regions and region sequences.

On the other hand, audio source state containing the analysis results, and audio modification state containing the user edits need to be persistent.

ARA model graphs may be rather huge when including the internal data on both sides, thus the ARA API was designed to deal with large archives efficiently. To avoid high temporary memory peaks, the data is not necessarily transmitted in one large data block. Instead, ARA uses a simple stream-like API that allows to read/write data in smaller chunks.

The stream should be both read and written in a consecutive order whenever possible. In some cases however, repositioning is inevitable, so the host must support it.

Because of its size and complexity, loading and storing an ARA graph may take a noticeable amount of time. Therefore ARA allows for displaying a progress for both processes.

Note that because all model data is stored in the ARA model graph, the persistent data returned per plug-in instance via the companion API will be nearly empty when ARA is enabled. Those companion archives can contain UI settings needed to reload the interface, but no model data.

More details about persistency are provided in later chapters of this document: Managing ARA Archives lays out how to deal with versioning and migration of ARA plug-in archives, and Partial Persistency describes the ARA 2 enhancements that enable hosts to selectively store or restore specific document objects rather than full documents, allowing for features like drag and drop of ARA objects between different host documents.

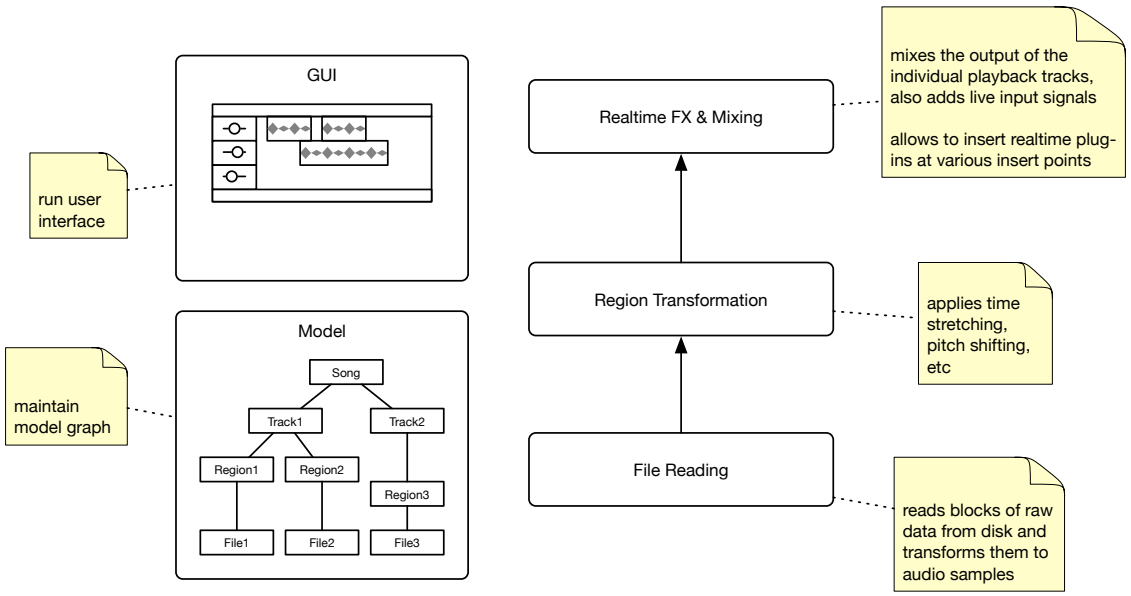
Host Signal Flow And Threading

A prerequisite for properly implementing ARA is understanding the signal flow in an ARA-enabled host. Before diving into the specifics of this, let's take a look at the typical playback signal flow in hosts without ARA.

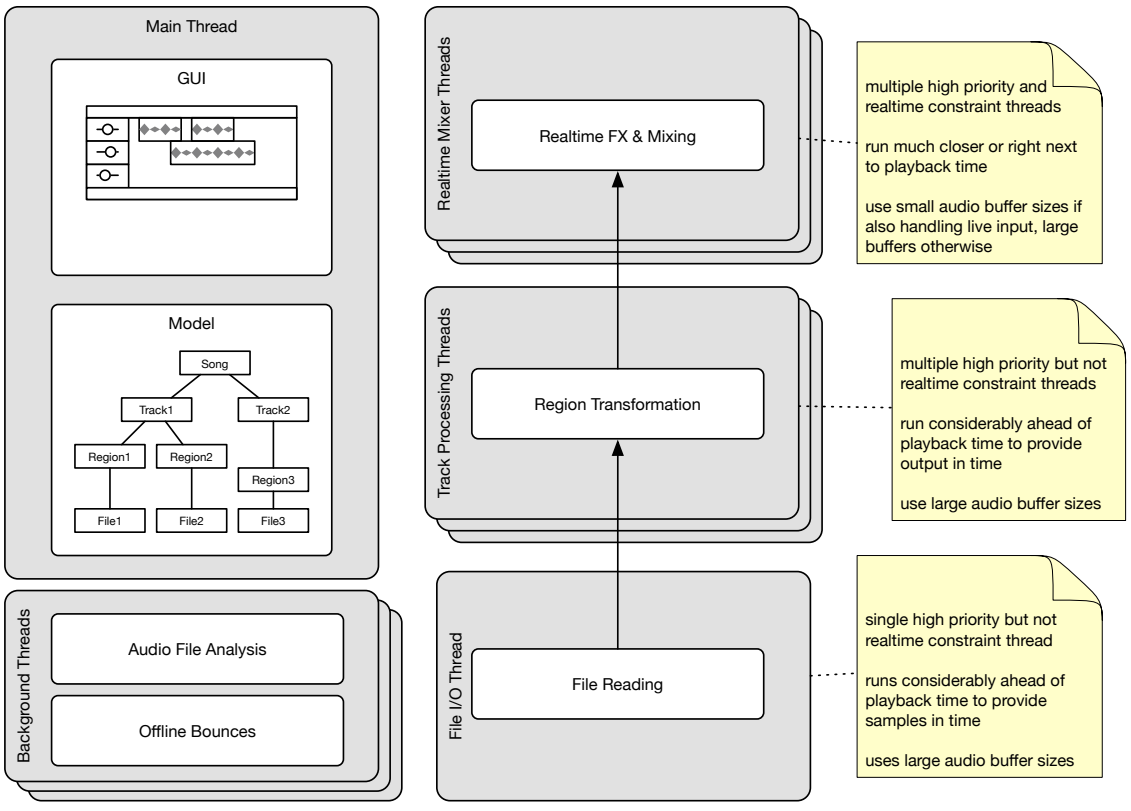
From a user's perspective, sections of audio files are arranged along a timeline. These snippets are usually called ”region” or ”clip” - we're using the term region here. The arrangement includes optional per-region transformations of the audio signal, such as level processing (region gain, fades), time stretching and pitch shifting. Additionally, sample rate conversion may need to be applied if the audio file does not match the playback sample rate.

The regions are typically grouped onto playback tracks, and the resulting track outputs are then fed into a realtime effects processing and mixing engine.

The following diagram illustrates a typical host implementation of these processing steps:



To optimize performance, the processing is usually distributed across a variety of threads with different constraints. The actual implementation differs widely between hosts, but typical host architectures assign threads according to the tasks outlined above:



All user interactions are processed on the main thread, and the model graph is maintained there accordingly. The graphical user interface is also updated from this thread.

If the host needs to execute heavy-weight calculations such as the analysis of audio material or an

”offline bounce”, it may offload these operations onto background worker threads to keep the main thread and user interface responsive.

The main thread will interrupt these low priority threads as needed to establish a fluid user experience. It depends on the actual host implementation which of tasks are scheduled on the main thread and which are executed by background threads.

The CPU critical realtime audio processing during playback is normally handled by a set of dedicated high priority render threads. These render threads may interrupt all other threads at any point, but the render calls of each plug-in instance are only called from a single render thread at any time.

In order to reliably meet their realtime constraints, these render threads must adhere to appropriate coding practices - most notably they must not be waiting for any shared resources such as system memory allocations.

To achieve low latency when processing live input signals, small audio buffers must be used and the rendering must happen as close as possible to the current playback time position.

Playback signals on the other hand can be rendered with much larger buffers and ahead of time, which considerably decreases CPU usage and makes the processing less vulnerable to dropouts. In fact, they can be rendered so early that they may not even need to obey strict realtime requirements.

Host accordingly tend to split the rendering between two groups of rendering threads: the semi-realtime playback track processing threads and the realtime effect processing and mixing threads.

In addition to these rendering threads, hosts typically create another thread dedicated entirely to load the audio file data from disk as needed for rendering.

This thread has to know in advance what data will be needed for rendering ”soon”, so that it has enough time to fetch it. This information is typically provided by some sort of sequencer that parses the model and handles the timely scheduling of such audio I/O requests. The scheduling may be executed on a dedicated thread or integrated into the track processing threads.

Inserting ARA Into The Signal Flow

While common plug-ins operate on blocks of realtime signal that can change arbitrarily between each render call, ARA-enabled plug-ins have no realtime inputs. Instead, they rely on random access to a conceptually static input signal (which can be deliberately updated by the host if needed.)

They are therefore perfectly suited to be processed by the track processing threads of the render engine instead of the ”traditional” realtime effect processing and mixing threads.

Depending on their capabilities and configuration, ARA-enabled plug-ins will replace some of the region transformations applied by the host, such as time stretching or sample rate conversion.

Since the region transformation performed by the ARA-enabled plug-in can include any arbitrary rearrangement of the original audio signal, the plug-in also is responsible for scheduling proper audio source sample reading ahead of time, while delegating the actual I/O operation to the host. The host will no longer stream the files through its playback engine.

Due to that setup, an ARA-enabled plug-in needs to be the first plug-in in any serial rendering chain. It is inserted before any other operation associated with the ”region”/”event” (such as fades, volume or mute) are applied. This may not be possible to easily implemented for all region operations that the host allows for (i.e. reversing the audio signal) - in that case, the affected operations must either be bounced to the audio source before adding the ARA plug-in, or disabled while ARA is in use.

As discussed above when introducing the ARA model graph, frequent updating audio source samples while ARA is in use should be avoided to prevent re-analysis and potential loss of any related user edits.

To that extent, it is not feasible to stack ARA effects without freezing/bouncing all but the top-of-stack effects.

The way ARA is injected into the signal flow also has implications on the bypass behavior. If a host bypasses the rendering of an ARA-enabled plug-in, it will need to take care of applying the time stretching as described in the playback region on its own in order not to mess up the timing when bypass is invoked.

This may or may not be possible depending on the host's capabilities. ARA-enabled plug-ins should thus consider following Melodyne's example and implement a dedicated ARA ”compare-with-original” mode. While in compare mode, Melodyne preserves the playback region time transformations, but bypasses any user edits stored in the audio modification. Accordingly applying the time stretching to the original audio source material allows the user to compare its edited version with the original in the context of the song.

In addition to their arrangement-based playback processing, ARA-enabled plug-ins can also create preview signals to guide users when editing the transformations in the UI. This could be playing a metronome click when editing timing while the host playback is stopped, or playing back the note currently dragged up and down by the user so that the current target pitch can be heard.

This obviously is a realtime process which needs to be handled in the mixer, not at track playback level. To allow optimizing for these different rendering tasks, separate render entities are needed. ARA 2 actively supports this by introducing explicit roles that hosts can assign to the companion API plug-in instances that perform the rendering.

Plug-In Instance Roles

Beside the two rendering duties discussed above, there are two more fundamental tasks that ARA-enabled plug-ins must implement: exchanging model data with the host, and providing a UI for the user to edit the plug-in specific arrangement transformations.

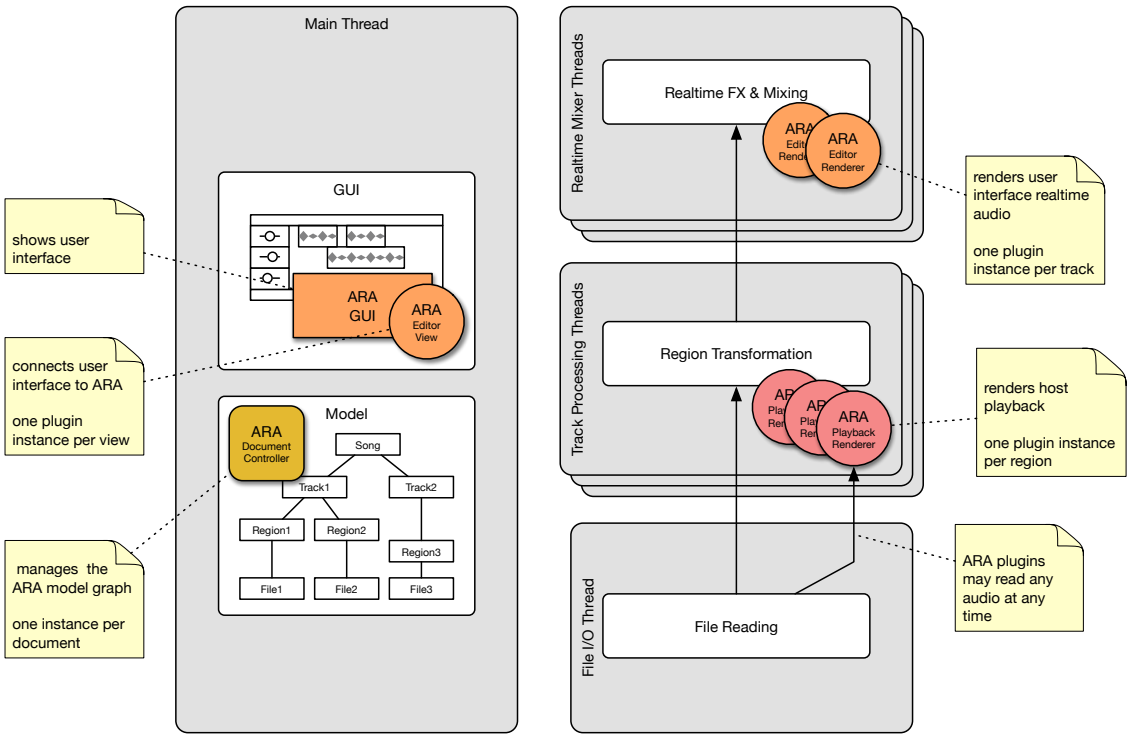
While the model is entirely ARA-specific and accordingly maintained through the ARA document controller, the rendering and the UI are implemented via companion API plug-in instances. These instances are toggled into ARA mode by ”binding” them to an ARA document controller and its associated model. When binding, the host also assigns the specific roles that it wants the ARA plug-in instance to assume:

- Playback Renderer
- Editor Renderer
- Editor View

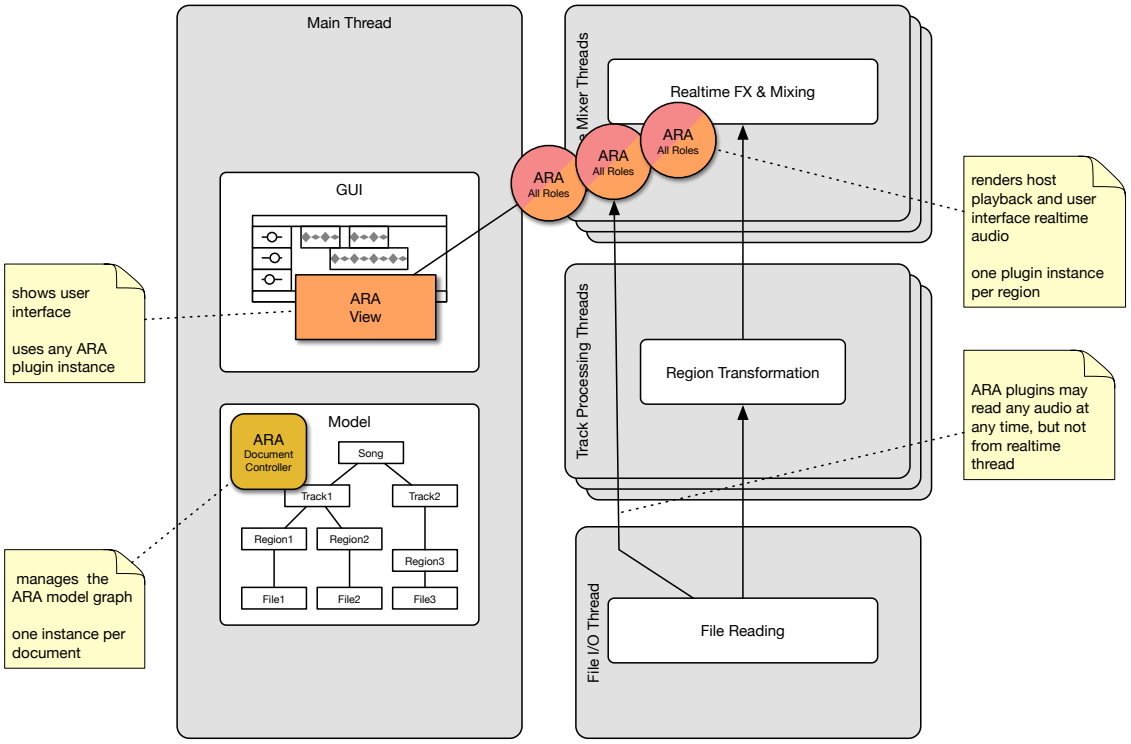
The roles also define the semantics of how to use the related companion API calls (see detailed API documentation).

For example, ARA playback renderes do not operate on realtime input - this is true both for the actual audio signal and for realtime parameter changes - since ARA is an offline API at its heart, any companion API parameters should not be used when ARA is active.

Depending on the host architecture, any plug-in instance may be assigend any combination of roles. A very natural setup keeps all three responsibilities completely separate, meaning each plug-in instance is assigned a single specific role. This is the case in the ARA implementation in Cubase/Nuendo:



Here's another example that closely resembles the original ARA 1 implementation in Studio One, where each plug-in instance assumes all three roles:



Comparing this host architecture to the previous one, the most important difference is that in the first architecture the playback rendering can be done in non-realtime, ”offline” mode, whereas the in the second

example it must be performed in realtime - ARA-enabled plug-ins need to deal properly with both modes.

Audio Access And Threading

A key feature of ARA is the multi-threaded, high-performance access from the plug-in to the samples of the host's ARAAudioSource. The API provides a synchronous (and thus potentially blocking) service that is well-suited for UI operations from the main thread or for background analysis jobs.

However these calls must not be used from realtime threads because they have an undefined execution time. Instead, plug-ins have to implement a non-realtime caching thread that fetches the audio source samples ahead of time so that they are available when the realtime renderer needs them, just like the host does internally for the non-ARA signal paths.

To ease the host implementation of the required multi-threaded audio access, ARA defines audio reader objects. Each of the audio readers may only be used by one thread at a time, so its interface is single-threaded. The plug-in will create as many audio readers as it has threads that need to read a given audio source concurrently.

Accessing the audio sources may not be possible at all times - for example when loading documents, the host may need to prepare some internal resources before samples can be provided. To manage such needs, the plug-in's document controller provides a synchronous call enableAudioSourceSamplesAccess().

While access is disabled, the plug-in must not use any of the audio readers. This specifically implies that when access is turned off, the plug-in must block the call until all reader threads have acknowledged this.

Implementing ARA

Preparing Your Implementation: Studying SDK Examples And Existing Products

Properly implementing ARA is typically a non-trivial task that requires careful preparation to be successful. Before starting out with designing and coding the ARA adoption of your product, we recommend that in addition to reading the ARA Design Overview, developers get familiar with existing ARA products to get a better feel of how ARA works from a users perspective.

Melodyne is a particularly useful example because it implements nearly all ARA features and provides extensive online documentation from a user's perspective `online here`. This documentation is not limited to plug-in features, but also shows how the various hosts interact with ARA plug-ins to provide a deep and unique user experience.

Next, we suggest to spend some time experimenting with the accompanying example code to see the API in action and get a feeling for the API and the typical call sequences.

The examples will also continue being useful during coding and testing - having full source code of both the host and the plug-in side examples allows to step through the flow of the program on both sides. Each developer can follow how their calls are typically processed on the other side of the API to catch any mismatches in the interpretation of the ARA model graph and its associated content.

Mini Host

The mini host example is a great starting point for understanding the basic management and lifetime of the ARA entities. It's written in pure C and walks very briefly through the core functionality of the ARA API. It is more educational than actually useful for debugging.

Test Host and Test Plug-In

We've also provided a much more elaborate test host which will put an ARA plug-in through various test scenarios, using typical host call sequences and patterns and responding to notifications from the plug-in.

Of course there's also a matching test plug-in which demonstrates how to iterate over host content, read and analyze audio source samples, and expose analysis results back to the host. It also handles persistence and demonstrates a playback renderer implementation. The test plug-in is accompanied by a ARATestChunkWriter tool for creating ARA audio file chunks for the plug-in.

Both the plug-in and host samples contain checks and assertions to ensure the other party plays by the rules, and should be used often to catch issues during the development process.

JUCE_ARA ARAPluginDemo

The ARATestPlugIn that ships with the ARA SDK already covers a wide range of ARA features, but is limited to console output and lacks a proper user interface. To address this, we've provided an additional experimental `JUCE_ARA plug-in demo` which is a plug-in with a UI that aims to visualize the ARA graph structure and its associated content in a representation that is close to actual ARA products.

Even without looking at the source code, this will be a great tool for host and plug-in developers alike to conveniently test how the ARA model graph and its associated content data will be interpreted by plug-ins and to experiment with the ARA UI integration features.

Since it requires using a custom fork of the extensive JUCE library, this demo is not directly included with the ARA SDK, but it contains the install_JUCE_ARA.cmake script to conveniently clone the corresponding Git repository, after which you can build and launch the Projucer to export a project for your IDE of choice via Open Example > Plugins > ARAPluginDemo.

Interaction With Existing Products

When developing and testing, it's also wise to regularly review your product's behavior with existing ARA enabled products, both from an end user's point of view and from a development perspective.

Plug-in developers should carefully use multiple ARA enabled DAWs when implementing and testing their products, as hosts often exhibit different usage patterns and call sequences that will need to be handled smoothly.

For host developers, Melodyne is a great benchmark of ARA feature functionality and should be loaded often to ensure a proper host implementation. Melodyne also supports ARA assertions, even in release builds, making it a great tool for detecting API errors.

Integrating The ARA SDK Into Your Products

While the ARA API has been defined in C for maximum compatibility, writing low level C code is not convenient. In order to ease ARA development and help creating robust plug-ins and hosts, the ARA library provides several layers of reusable C++ code on top of the underlying bare-bones C API.

Each layer of the ARA SDK provides its own level of abstraction and convenience. Picking the API level that's right for you is an important decision. The following overview of each layer shall help you with this task, as should the example code in `Examples` which illustrates how each layer is used in practice.

ARAInterface, Debug

The lowest level of abstraction is the ”raw” C API contract defined in `ARAInterface.h`. This layer is header-only and merely defines the C structures that are exchanged between the API partners. It does not contain any executable code, and the abstract definitions therein are designed to be as flexible and interoperable as possible to support a large variety of architectures, languages and compilers.

The accompanying `ARADebug.h` helpers provided with the SDK are compatible with this lowest C layer; they only add minimal standard C and OS dependencies and ease implementing ARA assertions.

The mini host example is written in C using the raw API and the ARADebug utilities to illustrate coding directly against this lowest layer of the SDK.

If you're not using C++, or are stuck with a very old C++ compiler, then this layer is your only choice. Otherwise, there should be only very few reasons to stick to this lowest level.

C++ Dispatcher, Utilities

The C++ dispatch level defines minimal abstract classes and helper templates for both hosts and plug-ins that encapsulate the underlying C ARA interface. It does not provide any validation or other functional code beyond forwarding the C API calls into the C++ world. Compared to ARAInterface, it merely adds dependencies to a few C++11 standard headers.

As an optional addition to the basic dispatching, `Utilities` provide various useful helpers designed to work in tandem with the dispatcher-based code.

The `Debug` folder also contains additional code for debugging when using these classes.

The dispatch level is typically used by host vendors, since the host code structures differ widely so that higher abstractions hardly apply.

The ARA test host makes use of the host dispatch classes as well.

For plug-in developers, starting at this layer may be convenient if the code base already contains C++ classes representing ARA objects, or if they find that the higher ARAPlug level has more abstraction than necessary or applicable.

ARAPlug

The ARAPlug layer provides a whole suite of C++ classes that represent every component of the ARA model graph, along with its document controller and representations of the ARA plug-in instance roles. It also provides many convenience classes and utility functions that simplify various tasks like archiving, content reading or sending update notifications to the host.

It is the highest in terms of abstraction and convenience and is used to for the ARA test plug-in included with the SDK. We recommend that new ARA plug-in developers start with this layer.

Please see the `ARAPlug` documentation for more information.

JUCE_ARA

JUCE is a popular C++ framework that is seeing wide-spread adoption in the audio developer community. Because our goal is to make ARA development easy and robust for as many developers as possible, we've teamed up with SoundRadix to created an experimental JUCE_ARA fork that drafts support for building ARA-enabled plug-ins with JUCE.

With version 7, the JUCE team has reviewed this fork and created a proper ARA integration in JUCE. The JUCE_ARA fork has been updated to use this official integration and now merely enhances the JUCE 7 ARAPluginDemo example with several more complex features which are beyond the scope of a simple example, but will be very useful when debugging ARA host applications. The fork also adds experimental implementations of draft API from the ARA SDK that is not reflected on the JUCE mainline yet. Note that Celemony is not endorsing the use of JUCE by providing this fork. The ARA integration in JUCE is merely an adapter, it does not provide any features that would be relevant when using ARA with a different framework. Consequently, the decision whether or not to use JUCE for any given project should be made independently of JUCE_ARA.

Mapping The Internal Model To ARA

The ARA model defines abstractions that allow for mapping a wide range of host and plug-in models to it. In several cases though, the mapping may not be entirely straight forward - there will likely be some ”impedance mismatches”.

The general guideline here should be for the host to configure and for the plug-in to interpret the ARA model not driven by their actual internal data model, but rather by the user representation thereof. This ensures that from a user's perspective, host and plug-in feel ”in sync” as tightly as possible.

Dealing With Overlapping Playback Regions

An example for this approach is dealing with region overlaps. Hosts typically define a z-order between regions, and depending on the host implementation and potentially user preferences, overlapping regions will sound concurrently, or only the top-most region will be audible in the overlap range (possibly using a cross fade range at the border where both the obscured and the top-most region are playing).

In the latter case, when configuring the ARA playback regions the host should restrict the borders of

the partially covered region(s) to those areas that are not fully obscured by other regions and thus actually do sound when playing back. Note that depending on the position and duration of the overlap(s), this may include slicing one partially covered region in the host into two or more regions at ARA API level. This way, the plug-in will only display and render audio material that is actually audible in the host.

Modelling Audio Modifications In The Host

Another area that tends to be no straight match is that hosts often either lack a dedicated audio modification abstraction, or use a somewhat different separation between playback region and audio modification.

If the modification abstraction is not part of the host model, then the typical approach is to use a fixed 1:1 relationship between audio source and audio modification, which results in any modification edit to be reflected in all playback regions that cover the edited modification range - the regions are ”aliases”, not distinct copies.

If the host model includes some concept of multiple modifications per audio file, there must be some logic implemented to decide whether a newly created region shall be using an existing modification, or whether a new modification should be created. In the latter case, there's the option to clone an existing modification or create a new one from the original audio source state.

To come up with a proper pattern to address those concerns in a given host, it is often valuable to compare this to the MIDI capabilities of the host as reference. A typical pattern is to create aliases per default, but provide an option to turn an alias into a distinct copy (modifier upon creation, or conversion command).

An alternate approach is to conceptually treat every region as an independent copy. In that case, the relationship between playback regions and modifications is 1:1. This pattern is especially viable if ARA's separation between modification and region does not map well to the host model.

Another consideration to keep in mind is that compared to plain audio files, ARA audio modifications are not limited in the time range they cover. Users can freely copy or move the original material around, this extends beyond start and end of the underlying audio source.

It is therefore desirable to allow ARA users to extend the playback region borders beyond the range of the audio source, instead of applying the typical border restriction implemented for regular audio files. Depending on whether this restriction is implemented at control layer or in the model, it needs to be considered either when mapping the model or when adjusting the user interface for ARA, see below.

Choosing An Appropriate Region Sequence Representation

Region sequences typically map to arrangement tracks or lanes, but as with the other mapping topics discussed above this may not always be a direct match. For example, some hosts allow for storing several alternate versions of the arrangement per lane, and let the user pick one version that is currently editable and played back.

In order for each track version to preserve its distinct ARA edit state, hosts must use separate audio modifications per version (while still sharing the underlying audio sources). When duplicating versions, the affected audio modifications will be cloned.

In order to prevent inactive versions from cluttering the plug-in UI, hosts should only expose a single region sequence per versioned track, containing only the current version's playback regions. If the active version is switched, the region sequences remain as-is, but all the playback regions are replaced as needed.

A similar pattern applies when comping: there should be a single region sequence representing the resulting comp lane, holding playback regions based on the selection made on the per-take lanes. This may be different from some host implementations where the resulting comp is merely a UI entity, and the

internal engine just plays the selected regions from the take lanes.

Configuring The Rendering

Setting Up An ARA Playback Renderer

The setup process for an ARA-enabled plug-in instance is not much different from setting up a non-ARA plug-in instance, but there's some considerations worth mentioning, mostly related to the fact that the audio source replaces the realtime input of the plug-in.

The established plug-in APIs usually define two states for a plug-in: a setup state where certain configurations such as the maximum render block size may be changed, but no rendering may occur, and a render state where the configuration is fixed but rendering may occur.

In most APIs, changing the I/O configuration is restricted to the setup state. Assigning playback regions to an ARA playback renderer plug-in instance can be considered an I/O change too, and thus is always restricted to the setup state. (Note that this is different for editor renderers, see below.)

Since ARA playback renderers have no realtime inputs, it would seem appropriate to suppress these in the supported I/O configurations published through the underlying companion APIs. However, given that preview renderers do use their inputs, and that a plug-in instance can be both playback and editor renderer, the I/O configurations made available by the plug-in must not depend upon ARA being enabled or not or on the assigned ARA instance roles.

Instead, for plug-in instances that do only playback but no editor rendering, ARA establishes the rule that the main inputs are simply never used - plug-ins never read them, and hosts do not need to supply a meaningful signal.

Ignoring the realtime inputs also means that ARA playback renderers should not incur any processing latency, because it can be completely compensated for inside the plug-in. Plug-ins must make sure to report this correctly through the companion API, and hosts should read the latency only after establishing the ARA binding.

Further, being independent of realtime input means that ARA playback renderers can be processed ahead of the actual playback location using rather large buffers to reduce CPU load, because the larger latency introduced by this can be fully compensated for. Note that this also includes proper visual latency compensation as supported by `VST3`'s `IAudioPresentationLatency` or by `Audio Units`'s `kAudioUnitProperty_PresentationLatency`.

Using larger buffers is particularly useful when a plug-in internally applies some sort of transformation into the frequency domain, because it then needs to apply internal buffering for the transformation and will cause high spikes of CPU load in all render slices where it can process a full transformation buffer, but will hardly do any work in all other render slices. Using render slice sizes that are about as large as the plug-in's internal transformation buffer therefore leads to a much steadier overall CPU load.

It is therefore highly recommended to process ARA-enabled playback with render slice sizes between 1024 and 4096 samples because these sizes are typically used internally in frequency domain related DSP algorithms.

Following these considerations, a host might even decide to render ARA playback renderers inside its file I/O thread (much like a realtime MP3 decoder), instead of performing the rendering from the realtime audio processing thread. Doing this is certainly valid and only requires to properly flag non-realtime usage to the plug-in, using `VST`'s `kVstProcessLevelOffline` or `Audio Units`'s `kAudioUnitProperty_OfflineRender` etc.

Preview Rendering

In addition to the actual audio output signals, ARA allows for temporary, auxiliary signals to be produced while editing the data. These signals are audible clues when performing the model edits, aiding and speeding up the editing process. They typically are only generated when song playback is stopped, in order not to corrupt potential bounces. A prominent example is Melodyne playing back a note when it is grabbed with the mouse and dragged up and down, so its new pitch is immediately audible. Another of these preview features is that it allows for optionally playing back a chord when the user selects it in its chord track. It also features a metronome in its audio source tempo definition editor.

These three examples show that there generally are two classes of these signals: sounds that are associated with a given playback region or region sequence, which should accordingly be routed through the same effect chain as used during playback of that region or sequence respectively, and song-global sounds that are not associated with a particular mixer channel, such as the aforementioned chord track preview or metronome. Some hosts, most prominently Cubase, feature an explicit monitoring channel for such signals with separate routing capabilities.

ARA 2.0 defines a dedicated editor renderer role that enables the host to set up these scenarios as supported: preview renderers can be set up with a set of playback regions or region sequences defined by the host. For a song-global preview renderer, that set remains empty. Hosts shall set up the preview so that it is unambiguous, i.e. all the sets should be fully disjoint, and only one set should be empty (song-global renderer).

Due to their very different nature, editor renderers comply to a different set of rules compared to playback renderers. In terms of signal flow, preview renderers add their signal to any input signal that the host may provide - or if a given plug-in instance is both playback and editor renderer at the same time, to the playback renderer output.

Since any preview signal is only temporary, drop-outs are acceptable if it eases the implementation. Further, generating the signal is a task in the plug-in that is fully transparent to the host. Therefore, the responsibility to properly react to any ARA model graph edits or signal routing changes (expressed by modifying the set of playback regions or region sequences of a given editor renderer) in a thread-safe fashion is entirely placed on the plug-in - the host can make any of such changes without toggling the plug-in temporarily from render state to setup state, as would be required for playback renderers. An easy way to deal with that requirement on the plug-in side is to simply cease and later resume any preview rendering whenever such a change to the model or the routing occurs.

Conversion Between Audio Source Format And Song Playback Configuration

The audio signal in any given audio source does not necessarily have to match the settings that the user has configured for playback of the song. The channel count or spacial channel layout may be different, or the source might be recorded at a different sample rate. Accordingly, channel format conversion and/or sample rate conversion (SRC) algorithms must be applied when processing the audio.

Some hosts deal with this issue by converting all audio files to the desired format upon importing them into a project, but others keep the audio files in their original format and perform on-the-fly conversion during playback.

Per default, these hosts should provide the audio source signal in its original format to any ARA plug-in in the model graph. This minimizes any artifacts that the conversions introduce, ensuring plug-ins will have the highest quality signal available for analysis and therefore yield the best results. It also prevents potential re-analysis and the according potential loss of edit data when the playback configuration is changed, which is especially important when dealing with ARA audio file chunks.

Plug-ins must therefore implement an appropriate handling of audio sources with different sample rates or channel formats. Some plug-ins such as Melodyne are capable of integrating the SRC into their DSP algorithms, thereby reducing the overall artifacts. On the other end of the spectrum, some plug-ins may not implement SRC at all, instead choosing to render silence in case of a sample rate mismatch and informing

the user that any SRC must be explicitly applied in the host. Note that the host does not know nor need to care about the plug-ins SRC implementation, it will always present the situation ”as is” and rely on the plug-in to deal with it appropriately.

If the plug-in does implement SRC, users should be aware that non-ARA audio sources or audio sources processed by different ARA plug-ins will be converted using different algorithms. This can result in differing levels of quality, or even changes in phase in extreme cases, which is problematic in applications where phase alignment is crucial.

In such cases, it is up to the plug-in vendor to educate users about the issue. For example, the plug-in could detect whether SRC is required, and instruct the user to either add the plug-in to all phase-aligned recordings to achieve consistent conversion, or to explicitly convert them all in the host before proceeding, whatever yields the preferred results.

Playback Region Head And Tail Times

Companion APIs allow plug-ins to publish a ”tail time” that informs hosts of a signal that will be appended beyond the end of the input signal. The value of the tail time typically depends on the settings that the user has dialed in. While it defaults to 0, it may be several seconds long in some use cases.

ARA adopts this concept for each region. Since ARA is a random access API, it also extends this to include a matching head time before the start of the region. Since ARA has the entire model down to the samples available, the head and tail times may not only depend on the user settings, but also on the arrangement and the content of the audio files. If changes in any of that calculation parameters result in different times, the plug-in will signal that change through a playback region content update notification to the host.

When rendering playback regions, host must take the head and tail time into account to allow the playback renderers to generate proper output. Head and tail times are a prerequisite for the ARA'scontent based fades” feature, see below.

Dealing With Denormals

When processing audio samples, de-normalized floating point values may cause severe performance penalties. A common way to handle this is to switch off denormals in the CPU, as they are irrelevant for the audible results of the rendering. A plug-in that uses this technique must potentially perform the switch whenever the host calls into the plug-in, and must switch back before exiting. To avoid this considerable overhead, it is recommended that host applications generally turn off denormals, so that plug-ins do not need to perform this switching at the begin and end of each render call.

Caching Especially CPU-intense DSP

In some cases, the processing demand for certain ARA edits may be so expensive that it cannot be rendered in realtime. To still allow for proper realtime playback, the processing results (or some intermediate data) must be cached.

If such caches can be created within a reasonably short amount of time, the plug-in can manage this data in a typical system-wide cache folder that will keep recently used data around until a certain threshold is exceeded. When loading a project that references data no longer part of the cache, the plug-in will recreate the data during the unarchiving process (for which the host will provide a proper progress bar). This approach is implemented e.g. by Melodyne's polyphonic audio processing.

There are however rare cases when calculating the render output is so expensive that it will stall the system for several minutes per affected audio modification. In this scenario, the cached data should

preferably be kept alive as long as the project that uses the data is being worked on. This is best achieved by placing the cache inside the host's project folder, i.e. a folder specific to the host document that can be used by plug-ins when storing data.

Finding this folder can be done using special companion API functions. The specific details of these functions vary depending on the companion API being used, but for example PreSonus provides an `IContextInfoProvider` extension to VST3 that can be downloaded from their `Developer Website`.

Analyzing Audio Material

What Can Be Analyzed?

ARA plug-ins typically need some initial analysis phase to build an internal model of the content of a given audio source before users can edit the DSP accordingly. If parts of the model can be mapped to the ARA content types, then the plug-in may export the analyzed data to the host, allowing it to be used as analysis engine - the `ARAFactory` publishes a list of content types that can be provided through analysis.

Note that any analysis performed by the plug-in may fail to provide meaningful results for a specific audio source and a given content type, e.g. if the signal is very noisy, or if it does not contain any information of that type. In this case, a plug-in may indicate that no content reading is possible, or it may fall back to some default state and keep providing ”initial grade” content despite an analysis was performed - the host must deal properly with both cases.

Manual Adjustments

Some plug-ins such as Melodyne allow for manual adjustments of the analysis results in their UI. When such edits happen, the host will be notified and content grade will be updated accordingly (see below).

In addition to the analyzable content, a plug-in may provide means for the user to define more content information for the audio source in its UI (e.g. specifying a key signature). While not determined through analysis and therefore not listed in the `ARAFactory`, such data will still be communicated to the host after the user has edited it, and the host can sync accordingly - listening to all relevant content changes should therefore be done regardless of a given plug-in's analysis capabilities.

Triggering Explicit Analysis

Depending on the design of the plug-in, the analysis of a given content type may be started automatically by the plug-in when creating an audio source, or it may be postponed until the user explicitly triggers it on demand throughout its UI. In cases where a host wants to use the analysis capabilities of a plug-in without user interaction, it must therefore explicitly request an analysis. (Host developers can test both behaviors with the ARA Test Plug-In, see the `ARA_ALWAYS_PERFORM_ANALYSIS` define in `ARATestDocumentController.cpp`.)

If user interaction in the host is blocked while the analysis is running, progress information should be provided to the user - plug-ins must support this if their analysis is taking a non-trivial amount of time.

In addition to progress information, a call is available to determine whether or not the analysis for a given content type is still in progress. This call is particularly relevant if the user can perform edits in the plug-in UI while the host waits for the analysis results - it may be possible for the user to restart the analysis

with different settings there.

Algorithm Selection

Plug-ins often provide sets of pre-configured analysis and other processing parameters configured for specific materials, such as distinguishing between percussive material versus tonal sounds with clearly perceived pitches. ARA 2.0 allows for exporting these of algorithms to the host so it can configure this appropriately when adding a new audio source - either through UI or through context, such as when recording a new take of a piece of music that has already been analyzed before.

Utilizing Content Exchange

Content information exchange is a key component of the relationship between ARA host and plug-in, and properly utilizing this information can yield powerful results. Content information resulting from plug-in analysis can be used by the host to adjust project settings to match the content, and plug-ins can use information provided by the host to adjust their transformation accordingly. Also, any information readily available in the host does not need to be analyzed by the plug-in.

Content information shared between plug-in and host will be flagged with a ”content grade” to indicate it's quality. It will be evaluated when deciding about how to use the information received from the API partner, as illustrated in the examples below.

When implementing content readers, it is important to be aware of potential rounding issues when converting between internal data structures and the continuous time used in the ARA API - both when calculating time stamps and when evaluating the optional content time range.

Musical Timing Information

A proper description of the musical timing (tempo and bar signatures) and how it changes over time is the foundation of many musical applications. Uses for this data range from presenting a natural editing grid to applying quantization or aligning and time-stretching some already existing recording to fit into some different piece of music.

Most hosts define a project-wide musical timing to which the entire arrangement is optionally aligned to. They typically provide a sophisticated UI to edit this timing. In ARA, this data is modeled as part of the musical context, and plug-ins can read it from the host on demand.

In order to align any audio signal musically meaningfully within this arrangement, the musical timing of the underlying audio source must also be known. If audio material is recorded in the host, the musical timing at the time of the recording should thus be stored alongside the audio.`If` pre-recorded audio material is imported into the host, it may also contain proper timing information. Otherwise, the host may prompt the user to specify the tempo and bar signatures manually, or it may implement automatic tempo and bar signature analysis.`The` latter option is non-trivial and involves very elaborate DSP, so it may be desirable to delegate this implementation to specialized ARA plug-ins such as Melodyne - the host can test if this feature is available in a given plug-in by evaluating the its `ARAFactory::analyzeableContentTypes`.

To start the analysis process, the host would first request the analysis of the audio source, then wait until the plug-in acknowledges the completion of the analysis. After checking that the analysis was successful, the host can then read the content information from the plug-in.

Content Grade Examples

After the plug-in has analyzed a given audio source, the content grade of the plug-in data will typically be `kARAContentGradeDetected`, indicating that the data results from an automatic detection without further user interaction.

This means that the results may not be entirely reliable, since the user has not reviewed them yet - applying automatic transformations such as time stretching to make the audio align with the musical timing of the song may thus yield undesirable results. Some host (e.g. Studio One) will therefore suppress such automatic adjustments at this early point - if so, the host or the plug-in may choose to indicate that the relevant information is available for user review (e.g. Melodyne utilizes a flashing tempo text field for this).

Should the user then review and potentially modify the timing definition in plug-in's UI, the grade will transition to `kARAContentGradeAdjusted`. The plug-in will notify the host about this change and the host can update its representation and will now consider that data fully valid and use it to its full potential.

This user approval is not required to be done on the plug-in side - if the host features some user interface for specifying audio source tempo information, the user can make adjustments there and the host will communicate this to the plug-in so it can update, exactly mirroring the use case described previously.

This symmetry in using content readers also extends to hosts that feature built-in tempo detection - when the plug-in queries the timing of a given audio source, the host can return a data of with the detected grade. The plug-in can then decide whether this information is plausible or whether it should try to re-detect the tempo. If however the tempo was available in the host because the user entered it, the grade would be adjusted and and the plug-in would accept this and skip its built-in detection.

Once host and plug-in agree on the musical timing, the plug-in can be instructed to apply time stretching when rendering playback regions as detailed in the next section. The host can further implement other features based on this information, such as allowing to set the project time line to match the tempo of an audio source etc.

Note that the user may be able to further edit the audio source timing definition in either the plug-in or the host, so both sides should continuously listen to further change notifications from the other side.

There are two more content grades defined in the ARA API. First, if a content reader is requested before the initial analysis has completed, or if an analysis failed to provide meaningful results, the content information will receive a grade of `kARAContentGradeInitial` to indicate that the data is not actually related to the content but merely represents some reasonable default value (e.g. a tempo of 120 BPM).

Finally, there are scenarios where the audio sources have been carefully preprocessed by some content producer, as is the case with some pre-packaged content libraries. Here the content information can be assumed to be fully correct and not to be changed by the user, which is represented by the `kARAContentGradeApproved`.

The content grades apply universally to all content types described in the ARA API, not just to musical timing information. The musical timing content types however are the most widely supported types on both sides of the API and thus serve best as examples illustrating how their content grade affects the actual user workflow on both the host and the plug-in side.

Notes And How Playback Transformations Affect Content Data

Another example of utilizing content information is performing an ”Audio to MIDI” conversion. This is done by using notes detected by the plug-ins analysis to create a new MIDI clip in the host, representing the notes being played in the original audio source. This content type is referred to in code as

ARAContentNote. Like in the previous example, this content information will be given a grade depending on whether any notes were actually detected or what kind of adjustments the user has made on their own.

Note content serves well for illustrating the effects of the transformations applied by the plug-in at the different parts of the ARA model graph. At audio source level, the returned notes would describe what was played in the original recording. If the application needs access to this original data, it is available here.

When reading at audio modification level, the data would be still in the original timing context, but include all edits that the user has made when rearranging the audio data inside the plug-in. This may be the least valuable information, since these user edits where made within the playback context, and may not be meaningful when being used in the original audio source context.

Reading per playback region yields the notes as rendered when playing back the arrangement, so if MIDI export is for example implemented by dragging audio regions in the host to MIDI tracks, this level of content reading must be used. Further, some hosts optionally draw MIDI-like representations of this content information on top of any audio region that is being associated with an ARA plug-in that delivers note content information.

Chords, Key Signatures And Other Content Types

The third important group of content types is the related to musical pitches and the harmonies they form, most prominently the chord progression in musical timing, and the overall key the notes are sounding in, available through `kARAContentTypeSheetChords` and `kARAContentTypeKeySignatures` respectively.

If the host provides this information in its musical context, it can be used by plug-ins to adjust their pitch editing of notes according to the harmonic structure of the song. This powerful feature can be seen e.g. when using Melodyne in Studio One or Cubase - notes detected by Melodyne can be pitched so that they follow the key scale and/or the chord progression of the song, allowing the host's chord track to affect the audio transformations in Melodyne.

There are several more content types available that are no discussed in detail here, for example related to tuning/intonation. More content types are likely to be added as the ARA API evolves. See Content Readers And Content Events in `ARAInterface.h` for a complete list and detailed description of each type.

Manipulating The Timing

A key benefit of ARA compared to established plug-in APIs is that because the musical timing information is provided both on audio source level and for the entire song timeline, hosts can use any ARA plug-in that advertises the time-stretch `kARAPlaybackTransformationTimestretch` capability in their `ARAFactory::supportedPlaybackTransformationFlags` as a very flexible time stretch engine.

In the most simple use case such plug-ins can perform a classic, linear time-stretching whenever different different durations are provided for playback regions in playback vs. modification. This would ignore the musical synchronization capabilities of ARA, and would be a typical behavior for plain sample editors that do not use a musical representation internally.

Hosts that deal with musical timing information however can behave much smarter. They can optionally quantize both playback region placement and stretching so that the musical timing in the song and the audio source are appropriately musically synchronized. Note that if the tempo in either domain is not constant,

then the time stretching is likely non-constant across the duration of the playback region.

Consider the following example (each line represents a beat drawn in time-linear):

```
song timing:    |      |      |      |      |  (constant tempo)
source timing:     |     |     |   |   |       (tempo is faster in the second half)
```

If a constant stretch factor is applied, this prevents the beats from aligning. However by tracking the tempo relationship between the source and the song and accordingly stretching the source material more in the second half where its tempo is faster, perfect alignment is possible:

```
song timing:    |      |      |      |      |  (constant tempo)
equal stretch:  |       |       |     |     |  (still faster in the second half, by same ratio)
tempo-adjusted: |      |      |      |      |  (constant tempo)
```

Note that this example is simplified, real-world calculations cannot assume constant tempo for either timeline and also need to properly reflect potential offsets between the first beat of the song/the audio source and the start of the song/the audio source.

Practical use cases often are not as extreme as our contrived example, and depending on the music a ”perfect” sync may not be desired - it's a popular production workflow to add a slightly sloppy instrument to an otherwise fully quantized arrangement to make it more interesting and lively.

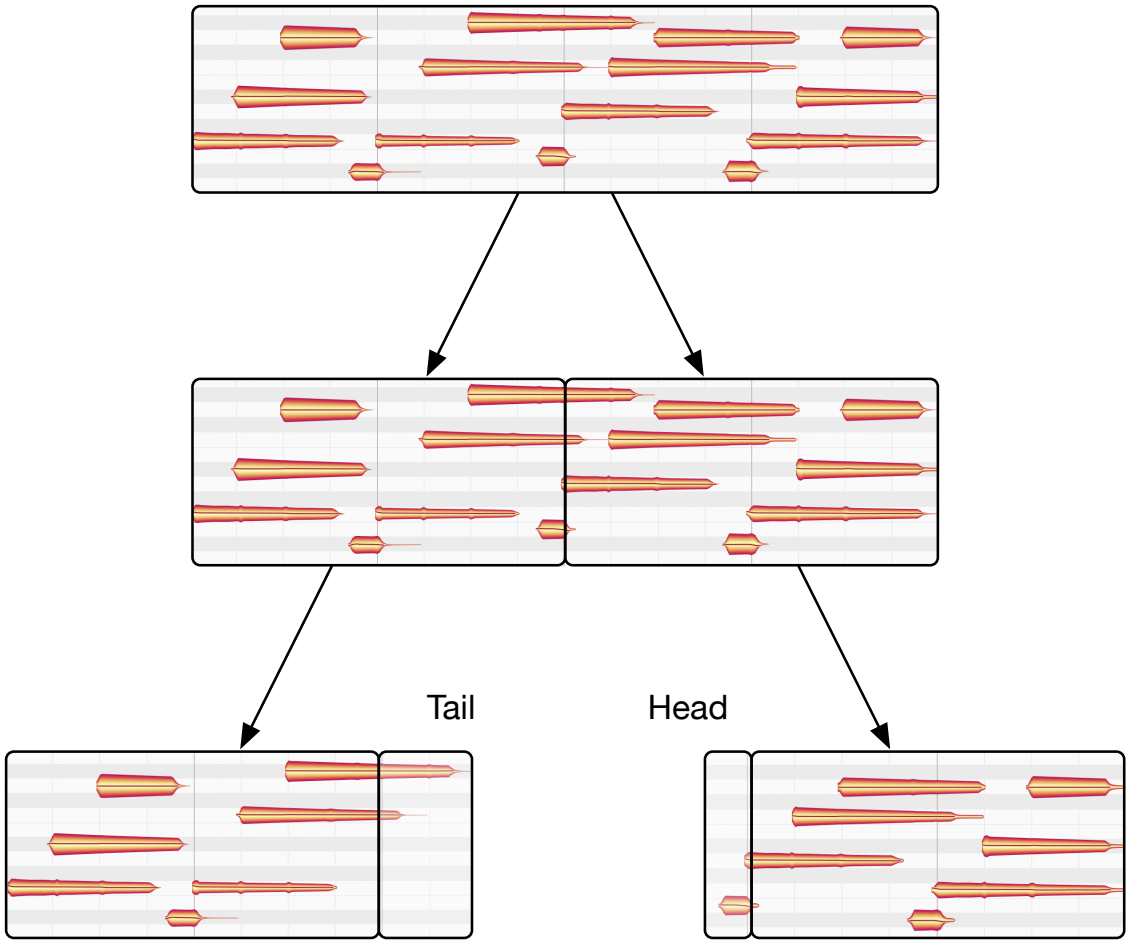
Hosts therefore typically allow users to choose between both mapping techniques, and they can communicate this choice to the plug-in by toggling the flag `kARAPlaybackTransformationTimestretchReflectingTempo` in the region's `ARAPlaybackRegionProperties::transformationFlags`.

Note that when mapping each individual note between both time domains, the plug-in may choose to apply any algorithm suitable. It may even offer the user an option to choose some sort of groove pattern that is to be taken into account when calculating the mapping. If a plug-in offers such an option, it must do so at audio modification level.

Finally, it should be pointed out that in addition to adjusting audio source data to the song timeline, the opposite feature may be desired as well: extracting the timing information from an audio recording and applying that to the song so that this audio recording becomes the new timing reference and other audio can align to it. This might for example be implemented by dragging playback regions onto the song timeline - the timing in the target ranges is replaced by the respective underlying audio source timing.

Content Based Fades

The head and tail time of playback regions can be leveraged by plug-ins to implement content based fades at the region borders. Instead of fading the overall signal, plug-ins can re-interpret these border fades depending on the type of content that will be playing when the fade occurs. This is best explained with an example:



This shows a region being sliced at a time that causes notes to be cut off or start after their transient. Instead of applying a traditional overall fade-out/fade-in, the plug-in could reflect this in its playback model so that each note that intersects with the transition is either completely played in the tail of the first or the head of the second region, so no fades are necessary at all.

This concept could be extended further - the plug-in could actually modify the notes that are intersected by the borders so that they start / stop as close as possible to the region boundaries, further improving the quality of the fade in a way that a traditional crossfade could not do. Head and tail time should always be kept at the minimum time that is required for achieving such a natural sounding processing of the region borders.

Content-based fades can be particularly strong when comping. Picking notes from either the first or the second region allows to deal with situations where overall fades are bound to create artifacts, such as when the notes in the second region are played slightly later than in the first.

The actual implementation of content based fades is entirely dependent on the internal processing algorithms of the plug-in. If those algorithms do not naturally provide an alternative to traditional overall fades, the plug-in should not implement this feature, and the host will then just use existing fade-based implementation used also for non-ARA regions.

Managing ARA Archives

An important aspect to consider when implementing persistency is dealing properly with the various persistent IDs in the ARA API and their associated versioning. There are three IDs involved: the

ARAFactory::factoryID, the ARAFactory::documentArchiveID and the plug-in ID as defined via the companion API.

The factoryID serves a runtime ID of the plug-in - it can be used to uniquely identify a specific version of a specific ARA plug-in, for example when copy/pasting plug-in state between documents. It also enables the host to determine whether the same ARA plug-in is installed across different companion APIs, such as an Audio Unit and VST3 version of the same plug-in. This allows to filter duplicate plug-ins and choose only a single companion API for each ARA plug-in as detailed later in Choosing Companion APIs.

As long as thefactoryID has not changed, the host-observable ARA behavior of the plug-in has not changed - e.g. it has the same analysis capabilities, supports the same archives etc. Hosts can therefore choose to store such information about the plug-in in a cache to avoid fully loading all plug-ins at each application startup, and update this cache whenever the factoryID changes.

The documentArchiveID uniquely identifies opaque archives of ARA plug-in state and must be stored by the host alongside the raw archive bytes whenever calling ARADocumentControllerInterface::storeObjectsToArchive() (or the legacy ARADocumentControllerInterface::storeDocumentToArchive()). The host should also store the user-readable meta information about the plug-in (ARAFactory::plugInName, ARAFactory::manufacturerName, ARAFactory::informationURL and ARAFactory::version) so it can later provide a proper error dialog should the archive be loaded on a system where the plug-in is missing or outdated, informing the user which version of which plug-in is needed and where to get it.

If a plug-in is updated in a way that makes its archived state incompatible with previous versions, both the factoryID and the documentArchiveID must be adjusted. Further, the previous documentArchiveID must be added to the list of ARAFactory::compatibleDocumentArchiveIDs and proper import code must be written to read existing archives in that previous format.

If the new version of the plug-in is installed, the host can detect that change via the updated factoryID. It will then read the new ARAFactory data and recognize the new documentArchiveID and the updated list of compatibleDocumentArchiveIDs. If a project is loaded that contains an old archive, the host will find the old ID in the list of compatible IDs and therefore knows that the plug-in will safely migrate the data.

A documentArchiveID is not necessarily unique to a specific plug-in. It is valid for any plug-in to be able to read or write archives that can also be processed by another plug-in. This allows for migration paths between different plug-ins, not just between updates of the same plug-in. It however also implies that the archive ID is not suitable to identify a given ARA plug-in - instead, the companion API provides proper IDs to handle that task.

To sum things up, let's step through the project load sequence from a host's perspective. The project archive typically contains a list of archived ARA document states, including the associated meta information from the ARAFactory described above as well as the companion API plug-in ID. For each of these states, the host first checks via the companion API whether the plug-in is currently installed in the system. If it is, the host then validates that the installed version can in fact read the archive by comparing its documentArchiveID and compatibleDocumentArchiveIDs with the document archive ID stored in the project. If ID is not listed, the installed plug-in is outdated and the host will show an appropriate error message.

If no plug-in has been found through the companion API, the host compares the project's document archive ID against the documentArchiveID and compatibleDocumentArchiveIDs of all installed ARA plug-ins. If another plug-in supports reading archives with the given IDs, the host can proceed to migrate the project to the new plug-in by loading the document archive via the new plug-in's document controller, and also switching all insert points in the arrangement that used the previous plug-in to the new one so that they can be bound properly to the migrated document controller.

With the ARA document controller and its associated document state restored, the host will then create companion API instances to fulfill the various required instance roles and bind them to the document controller. Both `kARAPlaybackRendererRole` and `kARAEditorRendererRole` are transient, so no state must be saved or restored for these via the companion API. Instances with the `kARAEditorViewRole` however may use that state to store customizable view configuration, so their state must be persisted and restored via the companion API.

Partial Persistency

Partial persistence is a feature added in ARA 2.0 that allows to limit storing or restoring objects to a subset of the current ARA graph by providing an optional filter to the store/restore operation. This can be used to export groups of objects from one document and import them into another - as an example, a dragged audio source from one document could be stored in a temporary archive and restored into a different document onto which it is dropped. Partial persistency also enables hosts to split the potentially large archive of a document into smaller chunks, which may be useful in data base or network sync contexts, and is used when reading ARA audio file chunks.

Copying ARA Data Between Documents

The most common use case for filtering the scope of archiving operations is implementing a feature that allows users to copy data between ARA documents, typically via copy/paste commands or drag and drop operations. This requires both an 'ARAStoreObjectsFilter' to store only the selected objects from the source document, and an ARARestoreObjectsFilter to initialize the newly created target objects.

When performing this operation, it is important that hosts manage the persistent IDs correctly, i.e. make sure that the IDs used in the source document while storing do not collide with IDs used in the target document. If there is a conflict, then hosts must map the stored IDs to new unique IDs and export this mapping to the plug-in in the 'ARARestoreObjectsFilter'. To see the filters in action, study the `testDragAndDrop()` function in the ARA Test Host.

Audio File Chunks

Another use case for partial persistence is importing WAV or AIFF files that contain ARA audio file chunks. These chunks allow for embedding the archive of an audio source state into the audio files that contain the audio samples for the source via the the `iXML standard`.

There are currently two main use cases for these chunks. First, for plug-ins which may require extensive analysis or manual editing of the audio source internal model, this is a way for content providers to ship audio content with these tasks already accomplished, so that user can go ahead and use the content with the supported plug-ins immediately without going to these time-consuming processes.
For example, 3rd party audio samples provider could create a loop library with polyphonic detection data for Melodyne and other plug-ins provided up-front in the iXML chunks, so user can load these loops and directly adjust the loops to their song's harmonies without further preparation.

The other major use case are plug-ins which generate audio content, e.g by de-mixing audio material or by exporting audio from one plug-in to another. An example here is SpectraLayers, which allows user to drag layers out of the plug-in into the host to process them separately from the remaining audio signal. The layer's samples are rendered into a new audio file, which is then passed to the host's file drop handler. The host imports the file and parses the iXML chunk to find that it should automatically add SpectraLayers

to the playback region created upon file import.

Note also that in both cases, the audio files are not created by the host but rather by code that is provided by the plug-in developer, either as part of the plug-in or as standalone application. The test plug-in from the SDK provides a dedicated ARATestChunkWriter tool for the plug-in, while Melodyne allows for storing such chunks from its regular standalone version.

To keep the chunks small and to speed up file loading, plug-in developers may accordingly choose to use a different archive format compared to ARA document archives, as done in Melodyne.

Companion API Considerations

Choosing Companion APIs

Supporting multiple companion APIs not only adds to the code which might introduce more bugs, it also adds another dimension to the already large test matrix for ARA enabled products and places a long-time maintenance burden on the developers.

To mitigate this effect as much as possible, it is strongly recommended that hosts which support multiple plug-in APIs when not using ARA limit this choice to only a single, preferably cross-platform companion API when implementing ARA.

This strategy requires on the other hand that plug-in developers should strife to implement ARA companion API support for all APIs that they support without ARA, enabling hosts to actually make the above restriction without excluding any specific plug-in from being used with ARA.

VST3: setActive() vs. setProcessing()

Both VST3 hosts and plug-ins should be aware of the difference between `IAudioProcessor::setProcessing` and `IComponent::setActive` when dealing with plug-in instances. As per the VST3 documentation, the `setProcessing` function should be limited to light-weight operations that reset the internal processing state, whereas `setActive` is used for heavy-weight initialization and memory allocations.

Unfortunately, when VST3 was initially established there where some confusions about these calls, which have resulted in some plug-ins not resetting properly upon `setProcessing`, and some hosts in turn calling `setActive` instead to work around issues with resetting plug-ins.

When adding ARA support, plug-ins are required to properly implement the VST3 specification, and hosts will accordingly rely on ARA plug-ins responding properly to `setProcessing`. Doing otherwise and sticking with `setActive` for reset can cause severe performance issues with large songs with several thousand playback regions, because accordingly as many playback renderer instances will be performing their memory allocations and complex initialization upon each reset, e.g. each when starting playback.

Audio Unit: Optimizing Buffer Allocation

The Audio Unit API uses a very complex I/O setup scenario with various connection options. While plug-ins must be prepared to handle all these options, hosts typically only use a specific one for all Audio Units. To allow optimization for the concrete connection in use, the Audio Unit specification includes the property `kAudioUnitProperty_ShouldAllocateBuffer` for both inputs and outputs.

Since ARA implementations typically rely on a high number of plug-in instances (typically one playback renderer instance per playback region, plus editor rendering instances), it is important that hosts enable this optimization by configuring the ARA Audio Unit instances accordingly.

Plug-in implementations must evaluate the property accordingly, which will happen automatically if

the implementation is based on the Core Audio Utility Classes (like done in the SDK test plug-in), but may require extra coding otherwise.

User Interface Considerations

For the better part of it, ARA relies on the companion API regarding how to create and use the graphical user interface of a plug-in. There are however a few things to consider in order to support the full ARA experience.

View Embedding

In many ways, ARA-enabled regions work similar to MIDI regions. It is therefore often desirable to embed the UI of an ARA plug-in into the main window, similar to the MIDI editors in many hosts. This is possible without further API changes across all currently supported companion APIs, and currently implemented e.g. in Studio One and Cubase.

Note that some hosts rely on custom extensions of the companion APIs for view embedding, for example Studio One - to indicate that view embedding is appropriate for any given plug-in (regardless of ARA), PreSonus has defined the custom, easy-to-implement `Presonus::IPlugInViewEmbedding` interface as part of their `PreSonus Plug-In Extensions` to VST3. This extension also been adopted by Cakewalk SONAR.

Reflecting Arrangement Selection In The Plug-In

Further following the MIDI editor analogy, those editors tend to optionally update the contents of their view based the selection in the arrange area. Since navigation and selection in the arrange area are core workflows that are well understood by all users, relying on them is preferable to implementing a different workflow for the MIDI editor.

The same considerations hold true for ARA-enabled plug-ins, so the ARA API provides means for the host to communicate the arrangement selection through the `ARAEditorViewInterface` interface. This selection optionally covers regions, tracks and a time range, so that the various arrangement selection techniques in current hosts can be mapped to a proper ARA 2 representation, enabling plug-ins to follow them in whatever is a suitable way for their UI.

Note that most MIDI editors include the option to ”pin” the current view state - ARA plug-ins are expected to implement such a feature too if applicable to their user interface.

Windows High DPI View Scaling

The Windows operating system allows for a per-application scaling of the UI through its `High DPI API`. To allow proper scaling of plug-in UI in host windows, ARA 2 plug-ins must support the associated view scaling provided through the various companion API, such as VST3's `IPlugViewContentScaleSupport` interface.

Key Event Handling

An area that is often problematic when using plug-in views via the established APIs is the routing of key commands. While ARA doesn't actually need to define anything special here, all developers shall be reminded to check proper routing of all key events to the plug-in depending on whether or not it has keyboard focus.

In your `Cocoa` implementation in particular, double-check that you've implemented all methods that handle events properly (most notably `-performKeyEquivalent:`) and that you're maintaining the first responder properly - see `Apple's documentation` on the topic.

Future ARA Development

There are several features that have been discussed when developing the API in its current form which were postponed for the time being because defining and implementing them properly needs more input from a wider range of development partners. The goal was to avoid designing API that would not actually be used, or would not work as desired in the various use cases that potential ARA development partners would come up with. As both hosts and plug-ins evolve and more and more developers are adopting the ARA standard, these features may shape up and eventually become part of the next generation of ARA:

- undo/redo integration with the host
- support for instrument plug-ins (i.e. no audio sources)
- support for MIDI plug-ins
- improved support for ”warp-”/”bend-”markers as found in many hosts
- support for phase-locked editing across concurrently recorded audio sources
- explicit support for comping/takes
- running the analysis while recording the audio
- optional side-chains and realtime input

Developer's feedback about how to design these or other missing features is highly welcome.

Use Cases and Testing

Synopsis

A major challenge when adopting an API as flexible and complex as ARA is understanding the potential use cases that will be encountered across all the possible host/plug-in combinations, and to design and test implementations accordingly.

This chapter describes various use cases that each host and plug-in should be aware of in order to build high quality ARA integrations. Intense quality insurance is key to a stable and thriving overall ARA platform and great user experience - passing this checklist as applicable is therefore also a prerequisite for Celemony granting developers the right to use its ARA Audio Random Access trademark for marketing a ARA-enabled products as such.

As a living document, this chapter will be regularly updated whenever common issues in current implementations become apparent.

As both hosts and plug-ins have very different features and capabilities, many of the following use cases may not apply in a specific host/plug-in combination. For example, a restoration plug-in will typically not bother with musical timing, time-stretching or MIDI notes. Nevertheless, all features which make sense in the given context shall be properly supported by all hosts and plug-ins that pick up ARA.

While testing, developers should also utilize the ARA Test Host and ARA Test Plug-In examples provided with the ARA SDK as well as Melodyne, which all feature extensive testing and validation as outlined in the previous chapter.

Render Timing

- Is sample timing correct in all cases? (It's easy to encounter an occasional single-sample offset due to rounding errors when feeding the plug-in's sample time)
- Is the latency of post-ARA realtime effects and audio hardware properly compensated for?
  - Is the playback still sample accurate across different tracks with different latencies?
  - For meters or playback location displays, is visual latency compensation working? (see VST3's `Steinberg::Vst::IAudioPresentationLatency` and AudioUnit's `kAudioUnitProperty_PresentationLatency`)
- If implemented in the host, do user-accessible track delay and/or region delay parameters still work properly when using ARA?

Musical Timing

- If both host and plug-in use musical song timing (tempo and signatures), is it properly exchanged? This can be tested both with Melodyne or the experimental `JUCE_ARA plug-in demo`.
  - Is the beat timeline drawn in the plug-in identical to the one drawn in the host?
  - Does the plug-in follow edits in the host timeline properly?
  - Does the grid align with the metronome/click track?
  - Can your implementation handle multiple tempo changes and multiple signature changes?
  - Is the musical timing always in sync when the latency compensation is active (both directions)?
- If both host and plug-in use musical timing for audio sources, is it properly synchronized in both directions?
  - If the host knows the file tempo and signature before the plug-in is added, does the plug-in read that information properly?

  - If not known before inserting, and the plug-in can analyze this, is the analysis result picked up by the host?
  - Does the plug-in update properly if the audio source tempo or signatures are edited in the host, and vice versa?

- If supported by the host, can audio source tempo and signature analysis provided by the plug-in copied to the song timeline? For example, Studio One implements this by letting users drag events (i.e. playback regions) to the timeline.

Time Stretching

- Does time-stretching work properly? Using an example audio file with a drum track that starts at 120 bpm then has a ritardando to 90 bpm:
  - can the user have this drum file analyzed in the plug-in and make it fit a song with constant 120 bpm? (As a host, have the audio analyzed by the plug-in, then read the timeline to find out the number of beats in the file, then in the host timeline pick a range with the same number of beats, then set up an according playback region and activate the TimestretchReflectingTempo flag
  - Starting from the results of the previous test, can the user stretch the file to halftime by either entering a stretch factor in the host or by dragging (if supported by the host)? Does the result sound correct in both cases?
  - Starting from the results of the first test, can the user use Melodyne's tempo dialog to re-interpret the tempo as halftime (so that the region/event is extended properly)?
  - Can the user do the opposite of the first test and make the song timeline fit the drum file? (As a host, have the audio analyzed by the plug-in, then read the timeline and copy the tempo curve to the song timeline, then set up a playback region that does not modify the file upon playback)

Signal Flow And Routing

- Are crossfades set in the host rendered correctly?
- Are content based fades rendered correctly (if implemented by the plug-in and supported by the host)?
- Does region/event solo/mute/volume behave as expected?
- Does track solo/mute/volume behave as expected?
- Is bypass behaving properly for ARA? (If the host chooses to bypass ARA plug-ins, it will need to apply any transformation done by the plug-in on its own - this may be tough esp. with regards to the the TimestretchReflectingTempo flag. The easier choice is to not implement bypass for ARA on the host side and instead leave that up to the plug-in.)

Maintaining The ARA Model

- Does undo/redo in the host work as expected if ARA-fied regions/events are involved? (Note: it may be necessary to keep some ARA objects around for Undo even if not used elsewhere, see deactivateAudioSource/ModificationForUndoHistory)
- If the host has a concept of alias versus copy of regions/events, do edits in the plug-in reflect accordingly across alias regions/events while not affecting copied regions/events?
- If the host has a concept of track versions/alternatives and supports multiple audio modifications per audio source, does it correctly create individual modifications per track version/alternative?

Audio-MIDI Conversion

- If implemented on both sides, does audio-to-MIDI work properly?
  - Is the timing of the MIDI 100% in sync with the audio?
  - Are the MIDI pitches correct? (It's easy to be off by an octave here because there are different interpretations between Yamaha and Roland of what pitch a ”C1” has)

Key Signatures And Chords

- Are enharmonic equivalents properly distinguished based on the circle of fifth index?
- Are the interval roles properly evaluated to distinguish e.g. from 2 from 9?
- If the name is provided or evaluated, are the music related unicode characters (sharp, flat, triangle etc.) properly handled?
- When reading chords, and the internal chord model is covering only a subset of the possible ARA chords, are there reasonable fallbacks for the cases not covered?

Persistence

- Can documents be archived and restored properly with all edits in tact (see also ARA Test Host `testArchiving()`)?
  - When doing a bounce, is the output identical before saving and after restoring?
- When restoring, does your host/plug-in fail gracefully if media files are missing (audio sources samples not available)?
  - Does the document recover properly if media files are re-assigned after they were missing upon restoring?
- Are archiving progress notifications being sent / received properly?
- Is drag/drop and copy/paste between documents properly including the respective ARA object state (see also ARA Test Host `testDragAndDrop()`)?
- Are ARA audio file chunks supported (see also ARA Test Host `testAudioFileChunkLoading()`)?
  - Audio files with chunks can be created by Melodyne 5 Standalone by storing detection data into the audio file. Is that detection data loaded properly when using Melodyne as ARA plug-in for such a file?
  - SpectraLayers 7 also creates ARA-enhanced files on demand when dragging layers out of the plug-in. These chunks request that the plug-in is opened automatically when the file is added, does this work?

Versioning

- On the host side, are the supported API generations evaluated properly?
  - Is there a proper error message for the user if a plug-in is either too old or too new?
- On the host side, are the versioned plug-in archive IDs properly stored and evaluated?

  - If a document is loaded which contains an incompatible plug-in chunk from a newer version of the plug-in, is there a proper error message containing the appropriate information about the newer plug-in? Is loading the incompatible chunk properly suppressed? This can be tested e.g. with Melodyne 5 vs. 4.

UI-Related Topics

- Does rendering in stop mode work if the UI is open?
  - Do editing operations in the plug-in (e.g. dragging a blob up and down in Melodyne) provide audible editor rendering results in stop mode?
  - Does the playback cursor in the plug-in follow the playback cursor in your host in stop mode?
- Does host playback control from the plug-in work work?
  - Are the cycle ranges properly synchronized between the plug-in and the host, with each side reflecting the changes on the other?
- Is key command handling working properly?
  - Are all key commands properly handled by the plug-in when its view has the focus (regardless of whether they are bound to menu items in the plug-in's menus or ”free” commands not bound to a menu)?
  - Are they properly ignored by the plug-in when has no focus?
  - Is there any conflicts when both the host and the plug-in implement the same key command, such as the standard key commands for copy/undo/paste or space bar for playback start?
  - Does it work across Windows, macOS, Linux (if applicable)?
- In the given host/plug-in scenario, does it make sense to optionally integrate the plug-in view into the main window like Studio One, Cubase and other hosts do with Melodyne?
- Does embedding the plug-in view into the host work fine in all places where it is implemented (separate window or embedded into a single-window-approach)?
  - Does resizing the view behave correctly?
  - Is repeatedly opening/closing views or switching between views of different plug-in instances working fine?
  - Does it work across Windows, macOS, Linux (if applicable)?
  - Is HiDPI supported properly?

General

- Does the implementation scale well both in terms of CPU and memory usage when being used with many long audio files at the same time (e.g. a real-life test scenario with a 30 minutes life-recording of 20+ tracks)?
- When reading UTF8-encoded strings from the other side, are any normalization requirements (e.g. for the text layout engine) properly enforced?
- Does the plug-in scan as an ARA enabled plug-in in the host?
  - For AudioUnit plug-in developers, is the `ARA”` tag” set in the `Info.plist` file?

Module Documentation

Basic Types

Detailed Description

Pre-defined types to ensure binary compatibility between plug-in and host. These types must be used when crossing the API boundary, but intermediate types can be used internally. For example, you can use your own struct representing color, but when defining color for ARA operations your internal color struct must be converted to ARAColor.

Fixed-size Integers

Detailed Description

ARA defines platform-independent signed integers with fixed size of 32 or 64 bits and for a pointer-sized signed integer.

Typedefs

- typedef uint8_t ARAByte

  Byte: 8 bits wide unsigned integer.
- typedef int32_t ARAInt32

  32 bits wide signed integer.
- typedef int64_t ARAInt64

  64 bits wide signed integer.
- typedef size_t ARASize

  Pointer-wide size value for ARA structs.

Boolean Values

Detailed Description

Since Microsoft still doesn't fully support C99 and fails to provide <stdbool.h>, we need to roll our own. On the other hand this ensures a fixed size of 32 bits, too. 32 bits were chosen so that ARABool is consistent with the other enum-like data types such as ARAContentType. Since ARABool is only used in temporary structs that are valid only for the duration of a call and likely passed in a register in most cases, there's no point in trying to optimize for size by using 8 bit boolean types. Note that in order to avoid conversion warnings in Visual Studio, you should not directly cast bool to ARABool or vice versa, but instead use a ternary operator or a comparison like this:

```
    00
araBool = (cppBool) ?  kARATrue :  kARAFalse;
cppBool = (araBool != kARAFalse);
```

Providing conversion operators for ARABool for C++ that handle this automatically is alas no viable option, because ARABool is only a typedef so this would lead to side-effects for all conversions from the integer type that ARABool is defined upon.

Typedefs

- typedef ARAInt32 ARABool

  Platform independent 32-bit boolean value.

Variables

- constexpr ARABool kARATrue { 1 }

  "true" value for ARABool.
- constexpr ARABool kARAFalse { 0 }

  "false" value for ARABool.

Enums

Detailed Description

ARA enums can either be used to represent distinct enumerations, or to declare C-compatible constant integer flags that can be or'd together as bit masks. To ensure binary compatibility between plug-in and host, the underlying type of ARA enums is always ARAInt32.

Macros

- #define ARA_32_BIT_ENUM(name) enum name : ARAInt32

  Define a 32-bit ARA enum type. The actual enum declaration is encapsulated in a macro to allow for adjusting it between C++, C and Doxygen builds.

Strings

Detailed Description

User-readable texts are stored as UTF-8 encoded unicode strings. It's not defined if and how the string is normalized - if either side has requirements regarding normalization, it needs to apply these after reading the string from the other side. Unicode rules apply regarding normalization, comparison etc. Both hosts and plug-ins are required to support at least all ISO/IEC 8859-1 based characters (from U+0020 up to U+007E and from U+00A0 up to U+00FF) in their text display rendering.

Typedefs

- typedef char ARAUtf8Char

  A single character.
- typedef const ARAUtf8Char * ARAUtf8String

  A string, 0-terminated.

Common Time-Related Data Types

Detailed Description

Some basic data types used in several contexts.

Typedefs

- typedef double ARATimePosition

  A point in time in seconds.
- typedef double ARATimeDuration

  A duration of time in seconds - the start of the duration is part of the interval, the end is not.
- typedef ARAInt64 ARASamplePosition

  Integer sample index, always related to a particular sample rate defined by the context it is used in.
- typedef ARAInt64 ARASampleCount

  Integer sample count, always related to a particular sample rate defined by the context this is used in.
- typedef double ARAQuarterPosition

  A position in musical time measured in quarter notes.
- typedef double ARAQuarterDuration

  A duration in musical time measured in quarter notes - the start of the duration is part of the interval, the end is not.

Sampled Audio Data

Detailed Description

The audio samples are encoded using these format descriptions. The data alignment and byte order always matches the machine's native layout.

Typedefs

- typedef double ARASampleRate

  Specified in Hz.
- typedef ARAInt32 ARAChannelCount

  Count of discrete channels of an audio signal. The spacial positioning of the channels may be provided via ARAChannelArrangementDataType.

Enumeration Type Documentation

ARAChannelArrangementDataType

```
enum ARAChannelArrangementDataType : ARAInt32
```

To avoid defining yet another abstraction of spacial layout information for the individual channels of an audio signal, ARA directly uses the respective Companion API's model of spacial arrangement. Since different Companion APIs are available, this enum specifies which abstraction is used.

Enumerator

kARAChannelArrangementUndefined	Used to indicate the feature is not supported/used (e.g. mono or stereo).
kARAChannelArrangementVST3SpeakerArrangement	For VST3, the channel arrangement is specified as Steinberg::Vst::SpeakerArrangement.

Enumerator

kARAChannelArrangementCoreAudioChannelLayout	For Audio Units, the channel arrangement is specified as the CoreAudio struct AudioChannelLayout. Note that according to Apple's documentation, "the kAudioChannelLayoutTag_UseChannelBitmap field is NOT used within the context of the AudioUnit." If possible, kAudioChannelLayoutTag_UseChannelDescriptions should also be avoided to ease parsing the struct.
kARAChannelArrangementAAXStemFormat	For AAX, the channel arrangement is specified as AAX_EStemFormat.

Color

Detailed Description

ARA color representation.

Classes

- struct ARAColor

  R/G/B color, values range from 0.0f to 1.0f. Does not include transparency because it must not depend on the background its drawn upon in order to be equally represented in both the host and plug-in UI - any transparency on either side must be converted depending on internal drawing before/after the ARA calls. More...

Class Documentation

struct ARAColor

R/G/B color, values range from 0.0f to 1.0f. Does not include transparency because it must not depend on the background its drawn upon in order to be equally represented in both the host and plug-in UI - any transparency on either side must be converted depending on internal drawing before/after the ARA calls.

Public Attributes

- float r

  red
- float g

  green
- float b

  blue

Object References

Detailed Description

ARA uses pointer-sized unique identifiers to reference objects at runtime - typical C++-based implementations will use the this-pointer as ID. C-style code could do the same, or instead choose to use

array indices as ID.

Those objects that are archived by the host can be persistently identified by an ARAPersistentID that the host assigns as a property of the object.

Persistent IDs

- typedef const char * ARAPersistentID

  Persistent object reference representation. Persistent IDs are used to encode object references between plug-in and host when dealing with persistency. Contrary to the user-readable ARAUtf8String, ARAPersistentIDs are seven-bit US-ASCII-encoded strings, such as "com.manufacturerDomain.someIdentifier", and can thus be directly compared using strcmp() and its siblings. They can be copied using strcpy() and must always be compared by value, not by address.

Markup Types

Type-safe representations of the opaque refs/host refs. The markup types allow for overloaded custom conversion functions if using C++, or for re-defining the markup types to actual implementations in C like this:

```
    00
#define ARAAudioSourceRefMarkupType MyAudioFileClass
#define ARAMusicalContextRefMarkupType MyGlobalTracksClass
```

... etc ...

- #define ARA_REF(RefType) struct RefType##MarkupType * RefType

  Plug-in reference markup type identifier.

  Examples:
  ARAMusicalContextRef
  ARARegionSequenceRef
  ARAAudioSourceRef
  ARAAudioModificationRef
  ARAPlaybackRegionRef
  ARAContentReaderRef
  ARADocumentControllerRef
  ARAPlaybackRendererRef
  ARAEditorRendererRef
  ARAEditorViewRef
  .

- #define ARA_HOST_REF(HostRefType) struct HostRefType##MarkupType * HostRefType

  Host reference markup type identifier.

  Examples:
  ARAMusicalContextHostRef
  ARARegionSequenceHostRef
  ARAAudioSourceHostRef
  ARAAudioModificationHostRef
  ARAPlaybackRegionHostRef
  ARAContentReaderHostRef
  ARAAudioAccessControllerHostRef
  ARAAudioReaderHostRef
  ARAArchivingControllerHostRef
  ARAArchiveReaderHostRef
  ARAArchiveWriterHostRef
  ARAContentAccessControllerHostRef
  ARAModelUpdateControllerHostRef
  ARAPlaybackControllerHostRef
  .

API Versions

Detailed Description

ARA implements two patterns for its ongoing evolution of the API: incremental, fully-backwards compatible additions by appending features to it versioned structs, and major, potentially incompatible updates through its API generations.

API Generations

Detailed Description

While purely additive features can be handled through ARA's versioned structs, ARA API generations allow for non-backwards-compatible, fundamental API changes. For hosts that rely on a certain minimum ARA feature set provided by the plug-ins, it also offers a convenient way to filter incompatible plug-ins. Plug-ins on the other hand can use the API generation chosen by the host to optimize their feature set for the given environment, such as disabling potentially costly fallback code required for older hosts when running in a modern host.

Macros

- #define ARA_DEPRECATED(generation)

  Markup for outdated API that should no longer be used in future development, but can still be supported for backwards compatibility with older plug-ins/hosts if desired.
  By defining ARA_ENABLE_DEPRECATION_WARNINGS as non-zero value it is possible to get deprecation warnings in the most common compilers, for inspecting deprecated API usage in a given project. These warnings are disabled by default in order to not interfere with code that supports older APIs.

- #define ARA_ADDENDUM(generation)

  Markup for struct elements which were added in later revisions of the API and may be omitted from the struct when dealing with older plug-ins/hosts.

- #define ARA_DRAFT ARA_ADDENDUM(2_X_Draft)

  Markup for draft API that is still under active development and not yet properly versioned - when using those struct elements, host and plug-in must agree on a specific draft header version!
  All uses of this macro will be replaced by ARA_ADDENDUM() upon final release. To quickly find all places in your project that use draft API, it's possible to temporarily redefine ARA_DRAFT to ARA_WARN_DEPRECATED(...).

Enumeration Type Documentation

ARAAPIGeneration

```
enum ARAAPIGeneration : ARAInt32
```

Enumerator

kARAAPIGeneration_1_0_Draft	private API between Studio One and Melodyne
kARAAPIGeneration_1_0_Final	supported by Studio One, Cakewalk/SONAR, Samplitude Pro, Mixcraft, Waveform/Tracktion, Melodyne, VocAlign, AutoTune
kARAAPIGeneration_2_0_Draft	supported by Studio One, Logic Pro, Cubase/Nuendo, Cakewalk, REAPER, Melodyne, ReVoice Pro, VocAlign, Auto-Align, SpectraLayers

Enumerator

kARAAPIGeneration_2_0_Final	most developers supporting kARAAPIGeneration_2_0_Draft are already working on updating to kARAAPIGeneration_2_0_Final (this is also required for ARM builds)
kARAAPIGeneration_2_X_Draft	reserved for future development

Versioned Structs

Detailed Description

In the various interface and data structs used in the ARA API, callback pointers or data fields may be added in later revisions of the current API generation. Each of these extensible structs starts with a structSize data field that describes how much data is actually contained in the given instance of the struct, thus allowing to determine which of the additional features are supported by the other side. All struct members that are later additions will be marked with the macro ARA_ADDENDUM. Members that are not marked as addendum must always be present in the struct. Accordingly, the minimum value of structSize is the size of the struct in the first API revision. When creating such a struct in your code, the maximum value is the size of the struct in the current API revision used at compile time. When parsing a struct received from the other side, the value may be even larger since the other side may use an even later API revision.

Note that when implementing ARA, it is important not to directly use sizeof() when filling in the structSize values. If you later update to newer API headers, the values of sizeof() will change and your code thus will be broken until you've implemented all additions. Instead, use the ARA_IMPLEMENTED_STRUCT_SIZE macro or similar techniques added in the ARA C++ library dispatcher code, see there.

Macros

- #define ARA_IMPLEMENTED_STRUCT_SIZE(StructType, memberName) (offsetof(ARA::StructType, memberName) + sizeof(static_cast<ARA::StructType *> (nullptr)->memberName))

  Macro that calculates the proper value for the structSize field of a versioned struct based on which features are actually implemented by the code that provides the struct. This may be different from sizeof() whenever features are added in the API, but not yet implemented in the current code base. Only after adding that implementation, the `memberName` *parameter provided to ARA_IMPLEMENTED_STRUCT_SIZE should be updated accordingly.*
  The ARA library C++ dispatchers implement a similar feature via templates, see ARA::SizedStruct<>.

- #define ARA_IMPLEMENTS_FIELD(pointerToStruct, StructType, memberName) ((pointerToStruct)->structSize > offsetof(ARA::StructType, memberName))

  Convenience macro to test if a field is present in a given struct.
  The ARA library C++ dispatchers implement a similar feature via templates, see ARA::SizedStructPtr::implements<>().

Debugging

Detailed Description

ARA strictly separates programming errors from runtime error conditions such as missing files, CPU or I/O overloads etc. Runtime errors occur when accessing external data and resources, which is always done on the host side of the ARA API. Accordingly, the host has the responsibility to detect any such

errors and to properly communicate the issue to the user. Since ARA leverages existing technologies, host implementations usually already feature proper code for this. With the error reporting done on the host side, plug-ins do not need to know any details about runtime errors - a simple bool to indicate success or failure is sufficient for implementing normal operation or graceful error recovery. Thus, ARA does not need to define error codes for communicating error details across the API. As an example, consider audio data being read from a server across the network - if the connection breaks, the host will recognize the issue and bring up an according user notification. If the plug-in requests the now inaccessible audio data, the host simply flags that an error occurred and the plug-in can either retry later or use silence as fallback data.

A different kind of errors are programming errors. If either side fails to properly follow the API contract, undefined behavior can occur. Tracking down such bugs from one side only can be difficult and very time consuming, thus ARA strives to aid developers in this process by defining a global assert function that both sides call whenever detecting programming errors related to the ARA API. When debugging (or when running unit tests), either side can provide the code for the assert function, so that no matter what side you're debugging from you can always inject your custom assert handling in order to be able to set proper breakpoints etc. The assert function is only a debug facility: it will usually be disabled on end user systems, and it must never be used for flow control in a shipping product. Instead, each side should implement graceful fallback behavior after asserting the programming error, e.g. by defining a special value for invalid object refs (NULL or -1, depending on the implementation) which will be returned as a placeholder whenever object creation fails due to a programming error on the other side and then filtering this value accordingly whenever objects are referenced.

Typedefs

- typedef void(∗ ARAAssertFunction) (ARAAssertCategory category, const void ∗problematicArgument, const char ∗diagnosis)

  Global assert function pointer. The assert categories passed to the global assert function are useful both for guiding developers when debugging and for automatic assert evaluation when building unit tests.
  The diagnosis text is intended solely to aid the developer debugging an issue "from the other side"; they must not be presented to the user (or even parsed for flow control). If applicable (i.e. if the category is kARAAssertInvalidArgument), the diagnosis should contain a hint about what problematicArgument actually points to - for example if a struct is too small, you'd pass the pointer to the struct along with a diagnosis message a la: "someExampleInterfacePointer->structSize < kExampleStructMinSize". Creating such appropriate texts automatically can be easily accomplished by custom assert macros.
  Finally, problematicArgument should point to the argument that contains the invalid data, so that the developer on that end can quickly identify the problem. If you can't provide a meaningful address for it, e.g. because the category is kARAAssertInvalidThread, pass NULL here.

Enumeration Type Documentation

ARAAssertCategory

`enum ARAAssertCategory : ARAInt32`

Hint about the nature of the programming error.

Enumerator

kARAAssertUnspecified	Not covered by any of the following codes.
kARAAssertInvalidArgument	Indicate that the caller passed invalid arguments.
kARAAssertInvalidState	Indicate that the call is invalid in the current state. e.g. if a document modification is being made without guarding it properly with ARADocumentControllerInterface::beginEditing() and ARADocumentControllerInterface::endEditing()

Enumerator

kARAAssertInvalidThread	Indicate that the call cannot be made on the current thread.

ARA Model Graph

Detailed Description

Document

Detailed Description

The document is the root object for a model graph and typically represents a piece of music such as a song or an entire performance. It is bound to a document controller in a 1:1 relationship. The document controller is used to manage the entire graph it contains. Because of the 1:1 relationship, the document is never specified when calling into the document controller. Edits of the document and any of the objects it contains are done in cycles started with ARADocumentControllerInterface::beginEditing() and concluded with ARADocumentControllerInterface::endEditing(). This allows plug-ins to deal with any render thread synchronization that may be necessary, as well as postponing any internal updates until the end of the cycle when the ARA graph has its full state available. A document is the root object for persistency and is the owner of any amount of associated audio sources, region sequences and musical contexts.

Plug-in developers using the C++ ARA Library can use the ARA::PlugIn::Document class.

Classes

- struct ARADocumentProperties

  Document properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere. More...

Class Documentation

struct ARADocumentProperties

Document properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.

- ARAUtf8String name

  User-readable name of the document as displayed in the host. The plug-in must copy the name, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a name for the document. In that case, the host should not make up some dummy name just to satisfy the API, but rather let the plug-in do this if desired - this way it can distinguish between a proper name visible somewhere in the host and a dummy name implicitly derived from other state.

Musical Context

Detailed Description

A musical context describes both rhythmical concepts of the music such as bars and beats and their distribution over time, as well as harmonic structures and their distribution over time. A musical context is always owned by one document. Musical contexts are not persistent when storing documents, instead the host re-creates them as needed.

Plug-in developers using the C++ ARA Library can use the ARA::PlugIn::MusicalContext class.

Classes

- struct ARAMusicalContextProperties

  Musical context properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere. More...

Typedefs

- typedef struct ARAMusicalContextRefMarkupType * ARAMusicalContextRef

  Reference to the plug-in side representation of a musical context (opaque to the host).
- typedef struct ARAMusicalContextHostRefMarkupType * ARAMusicalContextHostRef

  Reference to the host side representation of a musical context (opaque to the plug-in).

Class Documentation

struct ARAMusicalContextProperties

Musical context properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARAUtf8String name

  User-readable name of the musical context as displayed in the host. The plug-in must copy the name, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a name for the musical context (which is typically true for all hosts that only use a single context per document.) In that case, the host should not make up some dummy name just to satisfy the API, but rather let the plug-in do this if desired - this way it can distinguish between a proper name visible somewhere in the host and a dummy name implicitly derived from other state.
- ARAInt32 orderIndex

  Sort order of the musical context in the host. The index must allow for the plug-in to order the musical contexts as shown in the host, but the actual index values are not shown to the user. They can be arbitrary, but must increase strictly monotonically.
- const ARAColor * color

  Color associated with the musical context in the host. The plug-in must copy the color, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a color for the musical context.

Region Sequences (Added In ARA 2.0)

Detailed Description

Region sequences allow hosts to group playback regions, typically by ”tracks” or ”lanes” in their arrangement. Each sequence is associated with a musical context, and all regions in a sequence will be adapted to that same context. Further, all regions within a sequence are expected to play back through the same routing (incl. same latency), typically the same ”mixer track” or ”audio channel”. Regions in a sequence can overlap, and such overlapping regions will sound concurrently. A region sequence is always owned by one document, and refers to a musical context. Region sequences are not persistent when storing documents, instead the host re-creates them as needed.

Plug-in developers using the C++ ARA Library can use the ARA::PlugIn::RegionSequence class.

Classes

- struct ARARegionSequenceProperties

  Region sequence properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere. More...

Typedefs

- typedef struct ARARegionSequenceRefMarkupType * ARARegionSequenceRef

  Reference to the plug-in side representation of a region sequence (opaque to the host).
- typedef struct ARARegionSequenceHostRefMarkupType * ARARegionSequenceHostRef

  Reference to the host side representation of a region sequence (opaque to the plug-in).

Class Documentation

struct ARARegionSequenceProperties

Region sequence properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARAUtf8String name

  User-readable name of the region sequence as displayed in the host. The plug-in must copy the name, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a name for the region sequence. In that case, the host should not make up some dummy name just to satisfy the API, but rather let the plug-in do this if desired - this way it can distinguish between a proper name visible somewhere in the host and a dummy name implicitly derived from other state.
- ARAInt32 orderIndex

  Sort order of the region sequence in the host. The index must allow for the plug-in to order the region sequences as shown in the host, but the actual index values are not shown to the user. They can be arbitrary, but must increase strictly monotonically.
- ARAMusicalContextRef musicalContextRef

  Musical context in which the playback regions of the sequence will be edited and rendered. Note that when rendering the playback regions via any plug-in instance, the time information provided for this plug-in through the companion API must match this musical context.
- const ARAColor * color

Color associated with the region sequence in the host. The plug-in must copy the color, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a color for the region sequence.

Audio Source

Detailed Description

An audio source represents a continuous sequence of sampled audio data. Typically a host will create an audio source object for each audio file used with ARA plug-ins. Conceptually, the contents of an audio source are immutable (even though updates are possible, this is an expensive process, and user edits based on the modified content may get lost). An audio source is always owned by one document, and in turn owns any amount of associated audio modifications. Audio sources are persistent when storing documents.

Plug-in developers using the C++ ARA Library can use the ARA::PlugIn::AudioSource class.

Classes

- struct ARAAudioSourceProperties

  Audio source properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere. More...

Typedefs

- typedef struct ARAAudioSourceRefMarkupType * ARAAudioSourceRef

  Reference to the plug-in side representation of an audio source (opaque to the host).
- typedef struct ARAAudioSourceHostRefMarkupType * ARAAudioSourceHostRef

  Reference to the host side representation of an audio source (opaque to the plug-in).

Class Documentation

struct ARAAudioSourceProperties

Audio source properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARAUtf8String name

  User-readable name of the audio source as displayed in the host. The plug-in must copy the name, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a name for the audio source. In that case, the host should not make up some dummy name just to satisfy the API, but rather let the plug-in do this if desired - this way it can distinguish between a proper name visible somewhere in the host and a dummy name implicitly derived from other state.
- ARAPersistentID persistentID

  ID used to re-connect model graph when archiving/unarchiving. This ID must be unique for all audio sources within the document. The plug-in must copy the persistentID, the pointer may be only valid for the duration of the call.
- ARASampleCount sampleCount

Total number of samples per channel of the contained audio material. May only be changed while access to the audio source is disabled, see ARADocumentControllerInterface::enableAudioSourceSamplesAccess().

- ARASampleRate sampleRate

  Sample rate of the contained audio material. May only be changed while access to the audio source is disabled, see ARADocumentControllerInterface::enableAudioSourceSamplesAccess(). Note that the sample rate of the audio source may not match the sample rate(s) used in the companion plug-in instances that render playback regions based on this audio source - plug-ins must apply sample rate conversion as needed. However, if the sample rate is changed, plug-ins are not required to translate their model to the new sample rate, and may instead restart with a fresh analysis, causing all user edits applied at the previous sample rate to be lost.

- ARAChannelCount channelCount

  Count of discrete channels of the contained audio material. May only be changed while access to the audio source is disabled, see ARADocumentControllerInterface::enableAudioSourceSamplesAccess(). As with sample rate changes, plug-ins may discard all edits and start with a fresh analysis if the channel count is changed.

- ARABool merits64BitSamples

  Flag to indicating that the data is available in a resolution that cannot be represented in 32 bit float samples without losing quality. Depending on its internal algorithms, the plug-in may or may not base its decision to read either 32 or 64 bit samples on this flag, see ARAAudioAccessControllerInterface::createAudioReaderForSource().

- ARAChannelArrangementDataType channelArrangementDataType

  Type information of the data the opaque channelArrangement actually points to. Host shall use the data type associated with the Companion API that was used to create the respective document controller.

- const void ∗ channelArrangement

  Spacial arrangement information: defines which channel carries the signal from which direction. The data type that this pointer references is defined by channelArrangementDataType, see ARAChannelArrangementDataType.
  If channelCount not larger than 2 (i.e. mono or stereo), this information may omitted by setting channelArrangementDataType to kARAChannelArrangementUndefined and channelArrangement to NULL. The behavior is then the same as in hosts that do not support surround for ARA: for stereo, channel 0 is the left and channel 1 the right speaker.
  To determine which channel arrangements are supported by the plug-in, the host will use the Companion API and read the valid render input formats.

Audio Modification

Detailed Description

An audio modification contains a set of musical edits that the user has made to transform the content of an audio source when rendered by the ARA plug-in. An audio modification is always owned by one audio source, and in turn owns any amount of associated playback regions. Audio modifications are persistent when storing documents.

Plug-in developers using the C++ ARA Library can use the ARA::PlugIn::AudioModification class.

Classes

- struct ARAAudioModificationProperties

  Audio modification properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere. More...

Typedefs

- typedef struct ARAAudioModificationRefMarkupType * ARAAudioModificationRef

  Reference to the plug-in side representation of an audio modification (opaque to the host).
- typedef struct ARAAudioModificationHostRefMarkupType * ARAAudioModificationHostRef

  Reference to the host side representation of an audio modification (opaque to the plug-in).

Class Documentation

struct ARAAudioModificationProperties

Audio modification properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARAUtf8String name

  User-readable name of the audio modification as displayed in the host. The plug-in must copy the name, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a name for the audio modification. In that case, the host should not make up some name (e.g. derived from the audio source), but rather let the plug-in do this if desired - this way it can distinguish between a proper name visible somewhere in the host and a dummy name implicitly derived from other state.
- ARAPersistentID persistentID

  ID used to re-connect model graph when archiving/unarchiving. This ID must be unique for all audio modifications within the document. The plug-in must copy the persistentID, the pointer may be only valid for the duration of the call.

Playback Region

Detailed Description

A playback region is a reference to an arbitrary time section of an audio modification, mapped to a certain section of playback time. It is linked to a region sequence, which in turn is linked to a musical context. All playback regions that share the same audio modification play back the same musical content, but may adapt that content to the given section of the musical context and to the content of other regions in the same region sequence (see content based fades). Note that if a plug-in offers any user settings to control this adaptation (such as groove settings), then these settings should be part of the audio modification state, not of the individual playback regions. A playback is always owned by one audio modification, and refers to a region sequence. Playback regions are not persistent when storing documents, instead the host re-creates them as needed.

Plug-in developers using the C++ ARA Library can use the ARA::PlugIn::PlaybackRegion class.

Classes

- struct ARAPlaybackRegionProperties

  Playback region properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere. More...

Typedefs

- typedef struct ARAPlaybackRegionRefMarkupType * ARAPlaybackRegionRef

  Reference to the plug-in side representation of a playback region (opaque to the host).
- typedef struct ARAPlaybackRegionHostRefMarkupType * ARAPlaybackRegionHostRef

  Reference to the host side representation of a playback region (opaque to the plug-in).

Class Documentation

struct ARAPlaybackRegionProperties

Playback region properties. Note that like all properties, a pointer to this struct is only valid for the duration of the call receiving the pointer - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARAPlaybackTransformationFlags transformationFlags

  Configuration of possible transformations upon playback, i.e. time-stretching etc. The host may only enable flags that are listed in ARAFactory::supportedPlaybackTransformationFlags.
- ARATimePosition startInModificationTime

  startInModificationTime and durationInModificationTime define the audible audio modification time range. This section of the modification's audio data will be mapped to the song playback time range defined below as configured by the transformationFlags, including optional time stretching. See Manipulating The Timing for more information.
- ARATimeDuration durationInModificationTime

  See startInModificationTime.
  Plug-ins must deal with durationInModificationTime being 0.0.
- ARATimePosition startInPlaybackTime

  startInPlaybackTime and durationInPlaybackTime define the relationship between the audible modification time range and the song playback time in seconds as communicated via the companion API. Musical context content (such as kARAContentTypeTempoEntries) is also expressed in song playback time.
- ARATimeDuration durationInPlaybackTime

  See startInPlaybackTime.
  Plug-ins must deal with durationInPlaybackTime being 0.0.
- ARAMusicalContextRef musicalContextRef

  Musical context in which the playback region will be edited and rendered. Note that when rendering the playback region via any plug-in instance, the time information provided for this plug-in through the companion API must match this musical context.
  Deprecated
  No longer used since adding region sequences in ARA 2.0, which already define the musical context for all their respective regions. If structSize indicates that a sequence is used (i.e. when using ARA 2.0 or higher), this field must be ignored by plug-ins. Hosts are free to additionally set a valid musical context if desired for ARA 1 backwards compatibility, or leave the field uninitialized otherwise.
- ARARegionSequenceRef regionSequenceRef

  Region sequence with which the playback region is associated in the host.
- ARAUtf8String name

  User-readable name of the playback region as displayed in the host. The plug-in must copy the name, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a name for the audio modification. In that case, the host should not make up some name (e.g. derived from the audio source), but rather let the plug-in do this if desired - this way it can distinguish between a proper name visible somewhere in the host and a dummy name implicitly derived from other state.

- const ARAColor * color

  Color associated with the playback region in the host. The plug-in must copy the color, the pointer may be only valid for the duration of the call. It may be NULL if the host cannot provide a color for the playback region. In that case, the host should not make up some color (e.g. derived from the track color), but rather let the plug-in do this if desired - this way it can distinguish between a proper color visible somewhere in the host and a dummy color implicitly derived from other state.

Enumeration Type Documentation

ARAPlaybackTransformationFlags

`enum ARAPlaybackTransformationFlags : ARAInt32`

Playback region transformations. Plug-ins may or may not support all transformations that can be configured in a playback region. They express these capabilities at factory level, and the host must respect this. Also used in ARAFactory::supportedPlaybackTransformationFlags.

Enumerator

kARAPlaybackTransformationNoChanges	Named constant if no flags are set. If no flags are set, the modification is played back "as is", without further adoption to the given playback situation.
kARAPlaybackTransformationTimestretch	Time-stretching enable flag. If time-stretching is supported by the plug-in, the host can use this flag to enable it. If disabled, the host must always specify the same duration in modification and playback time, and the plug-in should ignore ARAPlaybackRegionProperties::durationInModificationTime.
kARAPlaybackTransformationTimestretchReflectingTempo	Time-stretching tempo configuration flag. If kARAPlaybackTransformationTimestretch is set, this flag allows to distinguish whether the stretching shall be done in a strictly linear fashion (flag is off) or whether it shall reflect the tempo relationship between the musical context and the content of the audio modification (flag is on).
kARAPlaybackTransformationContentBasedFadeAtTail	see ARAPlaybackTransformationContentBasedFades
kARAPlaybackTransformationContentBasedFadeAtHead	see ARAPlaybackTransformationContentBasedFades

Enumerator

kARAPlaybackTransformationContentBasedFades	Content-based fades enabling flags. These flags are used to enable smart, content-based fades at either end of the playback region. If supported by the plug-in, the host no longer needs to apply its regular overall fades at region borders, but can instead delegate this functionality to the plug-in. Based on the region sequence grouping, the plug-in can determine neighboring regions and utilize head and tail time to calculate a smart, musical transition. Even when no neighboring region is found, it may be appropriate to fade in or out to avoid cutting off signals abruptly. Note that while the transformation of a playback region can be defined separately for each border of the region, it must be enabled for either both or neither border in the ARAFactory::supportedPlaybackTransformationFlags.

Content Reading

Detailed Description

Content Updates

Detailed Description

There are several levels of abstraction when analyzing musical recordings. Initially, there is the signal in its ”physical” form. On the next level, the signal interpreted as a series of musical events - the notes played when creating the signal. These notes and their relationship in time and pitch can be interpreted further, leading to abstractions like tempo, bar signatures, key signatures, tuning and chords.

Updates may happen on any of these levels, both independently or concurrently. In the most simple but most unlikely case, the signal is completely replaced, and all the higher abstractions therefore also invalidated. This also means that all user edits done in an audio modification will be lost. More likely is a minor modification of the signal, such as applying a high pass filtering to remove rumble in the audio source. This will not change any higher abstractions (all notes etc. remain the same), so any edits inside the audio modification or any notation of the music based on the analysis will remain intact. Another case is the correction of the analysis by the user. The signal does not change in this case, but mis-detected notes are added or removed so the mid-level abstraction which is considered with the notes changes. Whether or not this also changes higher interpretations such as the detected harmonic structure depends on the case at hand.

ARA defines a set of flags that allow to communicate the level of change, which helps to avoid unnecessary flushing of the user edits inside an audio modification and allows for optimizations of the analysis. The flags are providing a guarantee what has NOT changed. This may seem odd at first but if for example a given host does not know about harmonies, it cannot make any assumption about whether these have changed or not.

Enumeration Type Documentation

ARAContentUpdateFlags

```
enum ARAContentUpdateFlags : ARAInt32
```

Flags indicating the scope of a content update. If notifying the API partner about a content update, the caller can make guarantees about which abstractions of the signal are unaffected by the given change. The enum flags describing these abstractions are or'd together into a single ARAInt32 value.

The C++ ARA Library encapsulates content updates in the ARA::ContentUpdateScopes class.

Enumerator

kARAContentUpdateEverythingChanged	No flags set means update everything.
kARAContentUpdateSignalScopeRemainsUnchanged	The actual signal is unaffected by the change. Note that in some cases even when the signal is considered to be unchanged, the values of the actual sample data may change, e.g. if the user applies a sample rate conversion.
kARAContentUpdateNoteScopeRemainsUnchanged	Content information for notes, beat-markers etc. is unaffected by the change.
kARAContentUpdateTimingScopeRemainsUnchanged	Content information for tempo, bar signatures etc. is unaffected by the change.
kARAContentUpdateTuningScopeRemainsUnchanged	Content readers for tuning (static or dynamic) are unaffected by the change. (added in ARA 2.0)
kARAContentUpdateHarmonicScopeRemainsUnchanged	Content readers for key signatures, chords etc. are unaffected by the change. (added in ARA 2.0)

Content Readers And Content Events

Detailed Description

Reading content description follows the same pattern both from the host and from the plug-in side. ARA establishes iterator objects called content reader to access the data in small units called content events. There are several types available, each defining a certain abstract representation of its associated events.

Upon creation, content readers are bound to a given content type and to an object of which the content shall be read. Optionally the reader can be restricted to only cover a given time range. Once created, its event count is queried and the individual events are read, then the reader is disposed of. This is all done immediately, reader objects are only temporary objects that are created and destroyed from the same stack frame.

The data pointer returned when reading an event's data remains owned by the content reader and must remain valid until the reader is either another event is read or the reader is destroyed.

The events returned by the reader are sorted in an order that depends on the content type, but generally follows their appearance on the timeline. If several events appear at the same (start-)time, their order is not defined and the receiver must apply further sorting if desired.

The C++ ARA Library offers convenient content reader classes for host and plug-in developers. Host developers can read plug-in content using ARA::Host::ContentReader, and plug-in developers can use ARA::PlugIn::HostContentReader to read host content.

Classes

- struct ARAContentTimeRange

  Content reader optional creation parameter: a range in time to filter content events. As an optimization hint, a content reader can be asked to restrict its data to only those events that intersect with the given time

range. Reader implementations should strive to respect this request, but focus on overall performance - the events actually returned may exceed the specified range by any amount, and calling code must evaluate the returned event positions/event durations.
Note that when calls accept a pointer to a content time range, that pointer is only valid for the duration of the call - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere. Further, in most of these calls the pointer to a content range may be NULL, indicating that the entire content range of the object should be read. More...

Typedefs

- typedef struct ARAContentReaderRefMarkupType * ARAContentReaderRef

  Reference to the plug-in side representation of a content reader (opaque to the host).
- typedef struct ARAContentReaderHostRefMarkupType * ARAContentReaderHostRef

  Reference to the host side representation of a content reader (opaque to the plug-in).

Class Documentation

struct ARAContentTimeRange

Content reader optional creation parameter: a range in time to filter content events. As an optimization hint, a content reader can be asked to restrict its data to only those events that intersect with the given time range. Reader implementations should strive to respect this request, but focus on overall performance - the events actually returned may exceed the specified range by any amount, and calling code must evaluate the returned event positions/event durations.

Note that when calls accept a pointer to a content time range, that pointer is only valid for the duration of the call - the data must be evaluated/copied inside the call, and the pointer must not be stored anywhere. Further, in most of these calls the pointer to a content range may be NULL, indicating that the entire content range of the object should be read.

Public Attributes

- ARATimePosition start

  Events at start time are considered part of the range.
- ARATimeDuration duration

  Events at start time + duration are not considered part of the range.

Enumeration Type Documentation

ARAContentGrade

```
enum ARAContentGrade : ARAInt32
```

Content grade: degree of reliability of the provided content information. The most prominent use of the content grade is to solve conflicts between data provided by the host and data found via analysis on the plug-in side. Another example is that when being notified about content changes in the plug-in, a host may choose to trigger certain automatic updates only if the grade of the content is above a certain reliability threshold.

Enumerator

kARAContentGradeInitial	Default data used as placeholder value. This grade can be used when no actual content information is present, such as the widely used default tempo of 120 bpm. This grade will typically be encountered while the analysis is still pending or still being executed. It may also be used if analysis failed to provide any meaningful results.
kARAContentGradeDetected	Data was provided by automatic content detection without any user intervention. Since the user has not reviewed the data, it may not be reliable in some cases.
kARAContentGradeAdjusted	Data was reviewed or edited by the user but not specifically approved as fully correct. This is the typical state of the data in the regular studio workflow.
kARAContentGradeApproved	Data has been specifically approved by the user as fully correct, e.g. for shipping it as part of a content library.

ARAContentType

`enum ARAContentType : ARAInt32`

Types of data that can be shared between host and plug-in.

Enumerator

kARAContentTypeNotes	Returns const ARAContentNote * for each note.
kARAContentTypeTempoEntries	Returns const ARAContentTempoEntry * for each tempo sync point.
kARAContentTypeBarSignatures	Returns const ARAContentBarSignature * for each bar signature change. In the original ARA 1 API, this value was called kARAContentTypeSignatures - while its name has been changed with ARA 2 to distinguish it from kARAContentTypeKeySignatures, its semantics and binary encoding are still the same.
kARAContentTypeStaticTuning	Returns single const ARAContentTuning *.
kARAContentTypeKeySignatures	Returns const ARAContentKeySignature * for each key signature change.
kARAContentTypeSheetChords	Returns const ARAContentChord * for each chord in a lead-sheet-like notation. (i.e. the sheet chord can imply notes that are not actually played in the audio, or vice versa. also, sheet chords are typically quantized to integer beats or even bars.)

Timeline

Detailed Description

ARA expresses musical timing as a mapping between song time measured in seconds and musical time measured in quarter notes. The mapping is created by dividing the timeline into sections of constant musical tempo. These tempo sections are then annotated as a list of tempo sync points, where each point represents both the end of one and the beginning of another section. The location of a tempo sync point is

specified both in song time and musical time. The actual tempo of a section can be easily derived from the relationship of the duration of the section in song time and the duration of the section in musical time (note that neither the quarters nor the seconds must necessarily be integer values here):

```
                      rightTempoEntry.quarterPosition - leftTempoEntry.quarterPosition
sectionTempoInBpm = ------------------------------------------------------------------ * 60.0 sec
                      rightTempoEntry.timePosition  - leftTempoEntry.timePosition
```

The advantage of providing such sync points whenever the tempo changes instead of specifying the tempo directly is that there are no rounding errors that sum up over time - whenever the tempo changes, this happens fully in sync. The disadvantage of this representation is that there is no way to express the tempo before the first and after the last tempo sync point, because the initial and final tempo sections stretch ”forever” into the past resp. future. ARA works around this by defining that the initial tempo is equal to the tempo between the first and the second tempo sync point and that the final tempo is equal to the tempo between the last-but-one and the last tempo sync point. This means that there must always be at least 2 sync points in a valid ARA time line definition.

To ease parsing the timeline, ARA further requires that there must be a tempo sync point given at quarter 0, even if there is no actual tempo change at this point in time. This allows for precisely determining any offset between time 0 seconds and quarter 0 without introducing possible rounding errors. (If a content range is specified, this only applies if quarter 0 is part of the content range.)

Musical timing is commonly not notated by simply counting quarter notes - instead bars are defined that form repeating patterns. ARA expresses this by providing a list of bar signatures. Like in standard musical notation, the bar signatures are expressed as a fraction of two integer values: numerator/denominator. The location of a bar signature is specified in musical time. To make sense musically, the distance between two bar signatures must be an integer multiple of the bar length of the earlier of the two signatures (even though the bar length itself may not be integer, e.g. when using a measure of 7/8). Note that when implementing the translation of these values to/from your code, potential rounding issues must be handled properly to ensure the desired positions are extracted.

Classes

- struct ARAContentTempoEntry

  Content reader event class: tempo map provided by kARAContentTypeTempoEntries. Event sort order is by timePosition. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader(). More...

- struct ARAContentBarSignature

  Content reader event class: bar signatures provided by kARAContentTypeBarSignatures. The event position relates to ARAContentTempoEntry, a valid tempo map must be provided by any provider of ARAContentBarSignature. Each bar signature is valid until the following one, and the first bar signature is assumed to also be valid any time before it is actually defined. The location of the first bar signature is also considered to be the location of bar 1. Event sort order is by position. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader(). More...

Class Documentation

struct ARAContentTempoEntry

Content reader event class: tempo map provided by kARAContentTypeTempoEntries. Event sort order is by timePosition. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader().

Public Attributes

- ARATimePosition timePosition

  Time in seconds relative to the start of the song or the audio source/modification.
- ARAQuarterPosition quarterPosition

  Corresponding time in quarter notes.

struct ARAContentBarSignature

Content reader event class: bar signatures provided by kARAContentTypeBarSignatures. The event position relates to ARAContentTempoEntry, a valid tempo map must be provided by any provider of ARAContentBarSignature. Each bar signature is valid until the following one, and the first bar signature is assumed to also be valid any time before it is actually defined. The location of the first bar signature is also considered to be the location of bar 1. Event sort order is by position. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader().

Public Attributes

- ARAInt32 numerator

  Numerator of the bar signature.
- ARAInt32 denominator

  Denominator of the bar signature.
- ARAQuarterPosition position

  Start time in quarter notes, see ARAContentTempoEntry.

Notes

Detailed Description

Notes in ARA correspond to what a composer would notate to describe the music. Notes are described by their position in time, their pitch and their relative volume. The pitch can be interpreted as frequency, but ARA also offers a musical description of the pitch very similar to MIDI: it defines the tuning for the overall musical scale and provides an integer number to identify the pitch for each note within this tuning, along with an average detune for each note actually played. ARA pitch numbers match MIDI note numbers, so that the note A4 has the value 69. This note is also used to specify the tuning reference, commonly at 440 Hz. At 440 Hz reference tuning the ARA pitch number 0 thus equals 8.1757989 Hz. Some notes may not have a well-defined pitch, such as percussive notes. For such notes, a frequency of kARAInvalidFrequency and a pitch number of kARAInvalidPitchNumber are used.

Classes

- struct ARAContentNote

  Content reader event class: notes provided by kARAContentTypeNotes. Event sort order is by startPosition. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader(). More...

Typedefs

- typedef ARAInt32 ARAPitchNumber

  Quantized pitch, corresponds to the MIDI note number in the range 0...127, but may exceed this range.

Variables

- constexpr ARAPitchNumber kARAInvalidPitchNumber { INT32_MIN }

  Used if there is no pitch associated with a note (e.g. purely percussive note).
- constexpr float kARAInvalidFrequency { 0.0f }

  Used if there is no pitch associated with a note (e.g. purely percussive note).
- constexpr float kARADefaultConcertPitchFrequency { 440.0f }

  Default tuning reference.

Class Documentation

struct ARAContentNote

Content reader event class: notes provided by kARAContentTypeNotes. Event sort order is by startPosition. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader().

Public Attributes

- float frequency

  Average frequency in Hz, kARAInvalidFrequency if note has no defined pitch (percussive).
- ARAPitchNumber pitchNumber

  Index corresponding to MIDI note number (or kARAInvalidPitchNumber).
- float volume

  Normalized level: 0.0f (weak) <= level <= 1.0f (strong). This value is scaled according to human perception (i.e. closer to a dB scale than to a linear one).
- ARATimePosition startPosition

  Time marking the beginning of the note (aka "note on" in MIDI), relative to the start of the described object (audio source/playback region).
- ARATimeDuration attackDuration

  Time marking the musical/quantization anchor of the note, relative to the start of the note.
- ARATimeDuration noteDuration

  Time marking the release point of the note (aka "note off" in MIDI), relative to the start of the note.
- ARATimeDuration signalDuration

  Time marking the end of the entire sound of the note (end of release phase), relative to the start of the note.

Tuning, Key Signatures and Chords (Added In ARA 2.0)

Detailed Description

ARA expresses "western standard" octave-cyclic, 12-tone scales as tunings and key signatures. While some applications such as Melodyne offer a much more complex model that allows for acyclic and/or micro-tonal scales, those models usually don't map well to each others, and introduce a complexity that can not meaningfully be handled by applications with the "main stream" model. Further, there is no standardized musical theory for expressing chords in such scales. Should the actual need to deal with more complex scales arise in the future, a new content type may be added to cover this.

Classes

- struct ARAContentTuning

  Content reader event class: periodic 12-tone tuning table provided by kARAContentTypeStaticTuning. Defines the tuning of each pitch class in the octave-cyclic 12-tone pitch system. Allows to import (12-tone) Scala files. Stretched tunings are not supported by ARA at this point, but may be added in a future release as an additional tuning stretch curve applied on top of this average tuning. ARA defines a single overall tuning (i.e. there's always only one event for this reader). As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader(). More...

- struct ARAContentKeySignature

  Content reader event class: key signature provided by kARAContentTypeKeySignatures. Defines the usage of each pitch class in the octave-cyclic 12-tone pitch system. This content type describes the key signatures as would be annotated in a score, not the local scales (which may be using some out-of-key notes via additional per-note accidentals). The event position relates to ARAContentTempoEntry, a valid tempo map must be provided by any provider of ARAContentBarSignature. Each key signature is valid until the following one, the first key signature is assumed to also be valid any time before it is actually defined. Event sort order is by position. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader(). More...

- struct ARAContentChord

  Content reader event class: chords provided by kARAContentTypeSheetChords. The event position relates to ARAContentTempoEntry, a valid tempo map must be provided by any provider of ARAContentBarSignature. Each chord is valid until the following one, and the first chord is assumed to also be valid any time before it is actually defined (i.e. its position is effectively ignored). The "undefined chord" markup (all intervals unused) can be used to express a range where no chord is applicable. Such gaps may appear between "regular" chords, or they can be used to limit the otherwise infinite duration of the first and last "regular" chord if desired. Event sort order is by position. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader(). More...

Typedefs

- typedef ARAInt32 ARACircleOfFifthsIndex

  The root of a key signature or chord as an index (or angle) in the circle of fifths from 'C'. Enharmonic equivalents such as Db and C# are distinguished:

- typedef ARAByte ARAKeySignatureIntervalUsage

  The ARAKeySignatureIntervalUsage defines whether a particular interval is used (kARAKeySignatureIntervalUsed) or not (kARAKeySignatureIntervalUnused). Future extensions of the API could further specify the usage of a given interval, similar to the chord intervals below. However since there are currently no clear-cut use cases for such a distinction, this is not yet specified.

- typedef ARAByte ARAChordIntervalUsage

  The ARAChordIntervalUsage defines whether a particular interval is used (kARAChordIntervalUsed) or not (kARAChordIntervalUnused), or if used may instead further specify the function of the interval in the chord by specifying its diatonic degree: 1 = unison, 3 = third, up to 13 = thirteenth Note that the bass note of a chord is treated separately, see below.

Markup values of ARAKeySignatureIntervalUsage.

- constexpr ARAKeySignatureIntervalUsage kARAKeySignatureIntervalUnused { 0x00 }

  Marks an interval of the ARAContentKeySignature as unused.

- constexpr ARAKeySignatureIntervalUsage kARAKeySignatureIntervalUsed { 0xFF }

  Marks an interval of the ARAContentKeySignature as used.

Markup values of ARAChordIntervalUsage.

```
common degrees per note if root is C:
  C           D           E     F             G                  A             B
  1    b9    2/9   #9/3    3    4/11   #11/b5   5    #5/b6/b13   6/7/13  7/#13   7
                                                                (7 only if dim)
```

- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree1 { 0x01 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree2 { 0x02 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree3 { 0x03 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree4 { 0x04 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree5 { 0x05 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree6 { 0x06 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree7 { 0x07 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree9 { 0x09 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree11 { 0x0B }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordDiatonicDegree13 { 0x0D }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used with the given diatonic function.
- constexpr ARAChordIntervalUsage kARAChordIntervalUsed { 0xFF }

  ARAChordIntervalUsage value when the corresponding chromatic interval is used, but its diatonic function is unknown.
- constexpr ARAChordIntervalUsage kARAChordIntervalUnused { 0x00 }

  ARAChordIntervalUsage value when the corresponding chromatic interval is not used.

Class Documentation

struct ARAContentTuning

Content reader event class: periodic 12-tone tuning table provided by kARAContentTypeStaticTuning. Defines the tuning of each pitch class in the octave-cyclic 12-tone pitch system. Allows to import (12-tone) Scala files. Stretched tunings are not supported by ARA at this point, but may be added in a future release as an additional tuning stretch curve applied on top of this average tuning. ARA defines a single overall tuning (i.e. there's always only one event for this reader). As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader().

Public Attributes

- float concertPitchFrequency

  Frequency of the concert pitch 'A' in hertz, defaulting to kARADefaultConcertPitchFrequency aka 440.0f.
- ARACircleOfFifthsIndex root

  Root key for the following per-key tunings.
- float tunings [12]

  Tuning of each note pitch as an offset from the equal temperament tuning in cent. Each entry defaults to 0.0f. The first entry relates to the root, increasing chromatically up to the full octave.
- ARAUtf8String name

  User-readable name of the tuning as displayed in the content provider. The tuning name may or may not include the root note, depending on context - for example, equal temperament does not care about the root note so it is always omitted, Werckmeister tunings typically imply a root note of C unless noted explicitly otherwise, etc. The name is provided only for display purposes, in case the receiver wants to display the tuning exactly as done in the sender instead of utilizing its built-in naming system for tunings based on the above properties. The receiver must copy the name, the pointer may be only valid as long as the containing ARAContentTuning struct. It may be NULL if the provider does not provide a name for the tuning in its UI.

Member Data Documentation

tunings `float ARAContentTuning::tunings[12]`

Tuning of each note pitch as an offset from the equal temperament tuning in cent. Each entry defaults to 0.0f. The first entry relates to the root, increasing chromatically up to the full octave.

```
Example:
Arabian Rast: {0.0f, 0.0f, 0.0f, 0.0f, -50.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -50.0f}
```

struct ARAContentKeySignature

Content reader event class: key signature provided by kARAContentTypeKeySignatures. Defines the usage of each pitch class in the octave-cyclic 12-tone pitch system. This content type describes the key signatures as would be annotated in a score, not the local scales (which may be using some out-of-key notes via additional per-note accidentals). The event position relates to ARAContentTempoEntry, a valid tempo map must be provided by any provider of ARAContentBarSignature. Each key signature is valid until the following one, the first key signature is assumed to also be valid any time before it is actually defined. Event sort order is by position. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader().

Public Attributes

- ARACircleOfFifthsIndex root

  Root key of the signature.
- ARAKeySignatureIntervalUsage intervals [12]

  Scales intervals (aka scale mode) of the signature. The index of this arrays entry is the chromatic interval to the keys root pitch class.
- ARAUtf8String name

  Optional user-readable name of the key signature as displayed in the content provider (including the root note name). Typically, the receiver has a built-in system to generate a suitable name for a key signature based on its internal abstractions. However, this internal model might cover all states that the ARA model can provide, or vice versa. If there is such a mismatch, the internal name generation algorithm will likely fail - in which case the receiver may fall back to the string provided here. Note that the utility library that ships with the ARA SDK contains C++ code that can create a proper name for the most common key

signatures, otherwise falling back to this name. The receiver must copy the name, the pointer may be only valid as long as the containing ARAContentKeySignature struct. It may be NULL if the provider does not provide a name for the key signature in its UI. When encoding the string to send it across the ARA API, flat and sharp symbols must be represented with the Unicodes 0x266D "MUSIC FLAT SIGN" and 0x266F "MUSIC SHARP SIGN" respectively (as also required for ARAContentChord::name).

- ARAQuarterPosition position

  Start time in quarter notes, see ARAContentTempoEntry.

Member Data Documentation

intervals `ARAKeySignatureIntervalUsage ARAContentKeySignature::intervals[12]`

Scales intervals (aka scale mode) of the signature. The index of this arrays entry is the chromatic interval to the keys root pitch class.

```
Examples (Hex Values):
major            {FF, 00, FF, 00, FF, FF, 00, FF, 00, FF, 00, FF}
natural minor    {FF, 00, FF, FF, 00, FF, 00, FF, FF, 00, FF, 00}
```

struct ARAContentChord

Content reader event class: chords provided by kARAContentTypeSheetChords. The event position relates to ARAContentTempoEntry, a valid tempo map must be provided by any provider of ARAContentBarSignature. Each chord is valid until the following one, and the first chord is assumed to also be valid any time before it is actually defined (i.e. its position is effectively ignored). The "undefined chord" markup (all intervals unused) can be used to express a range where no chord is applicable. Such gaps may appear between "regular" chords, or they can be used to limit the otherwise infinite duration of the first and last "regular" chord if desired. Event sort order is by position. As with all content readers, a pointer to this struct retrieved via getContentReaderDataForEvent() is still owned by the callee and must remain valid until either getContentReaderDataForEvent() is called again or the reader is destroyed via destroyContentReader().

Public Attributes

- ARACircleOfFifthsIndex root

  Root note of the chord.
  Examples: F: root = -1 C/E: root = 0 G/D: root = 1.

- ARACircleOfFifthsIndex bass

  Bass note of the chord. Usually identical to the root, but may be different from root, or even different from any other note represented by the intervals below.
  Examples: F: bass = -1 C/E: bass = 4 G/D: bass = 2.

- ARAChordIntervalUsage intervals [12]

  Chords intervals, defining gender, suspensions and extensions. The index of this array's entry is the chromatic interval to the chords root pitch class. Depending on the chord's musical interpretation, the interval corresponding to the bass note may or may not be included in here. If all intervals are unused, this chord represents an "undefined chord", which is used as a markup for gaps in the chord progression.

- ARAUtf8String name

  Optional user-readable name of the chord as displayed in the content provider (including the root note name). Typically, the receiver has a built-in system to generate a suitable name for a chord based on its internal abstractions. However, this internal model might cover all states that the ARA model can provide, or vice versa. If there is such a mismatch, the internal name generation algorithm will likely fail - in which case the receiver may fall back to the string provided here. Note that the utility library that ships with the ARA SDK contains C++ code that can create a proper name for any ARA chord, completely avoiding the necessity to use this name string. The receiver must copy the name, the pointer may be only valid as long as the containing ARAContentChord struct. It may be NULL if the provider does not provide a name for the

chord in its UI.
Chord annotation systems often use symbols in addition to Latin letters and Arabic numbers. Depending on UI considerations such as the font in use, different applications may use different Unicode codes to represent the same symbols internally. In order to send strings containing such symbol across the ARA API without ambiguities, they must be mapped to the following Unicode codes:

- ARAQuarterPosition position

  Start time in quarter notes, see ARAContentTempoEntry.

Member Data Documentation

intervals `ARAChordIntervalUsage ARAContentChord::intervals[12]`

Chords intervals, defining gender, suspensions and extensions. The index of this array's entry is the chromatic interval to the chords root pitch class. Depending on the chord's musical interpretation, the interval corresponding to the bass note may or may not be included in here. If all intervals are unused, this chord represents an ”undefined chord”, which is used as a markup for gaps in the chord progression.

```
Examples (hexadecimal values):
major            {FF, 00, 00, 00, FF, 00, 00, FF, 00, 00, 00, 00}
major            {01, 00, 00, 00, 03, 00, 00, 05, 00, 00, 00, 00}
minor            {01, 00, 00, 03, 00, 00, 00, 05, 00, 00, 00, 00}
major 13         {01, 00, 09, 00, 03, 00, 00, 05, 00, 0D, 07, 00}
major add13      {01, 00, 00, 00, 03, 00, 00, 05, 00, 0D, 00, 00}
major 6          {01, 00, 00, 00, 03, 00, 00, 05, 00, 06, 00, 00}
```

name `ARAUtf8String ARAContentChord::name`

Optional user-readable name of the chord as displayed in the content provider (including the root note name). Typically, the receiver has a built-in system to generate a suitable name for a chord based on its internal abstractions. However, this internal model might cover all states that the ARA model can provide, or vice versa. If there is such a mismatch, the internal name generation algorithm will likely fail - in which case the receiver may fall back to the string provided here. Note that the utility library that ships with the ARA SDK contains C++ code that can create a proper name for any ARA chord, completely avoiding the necessity to use this name string. The receiver must copy the name, the pointer may be only valid as long as the containing ARAContentChord struct. It may be NULL if the provider does not provide a name for the chord in its UI.

Chord annotation systems often use symbols in addition to Latin letters and Arabic numbers. Depending on UI considerations such as the font in use, different applications may use different Unicode codes to represent the same symbols internally. In order to send strings containing such symbol across the ARA API without ambiguities, they must be mapped to the following Unicode codes:

- flat symbol: 0x266D ”MUSIC FLAT SIGN”
- sharp symbol: 0x266F ”MUSIC SHARP SIGN”
- upward-pointing triangle used to annotate major 7 chords: 0x2206 ”INCREMENT”
- minus sign used to annotate minor chords: 0x002D ”HYPHEN-MINUS”
- circle with slash used to annotate half-diminished chords: 0x00F8 ”LATIN SMALL LETTER O WITH STROKE”
- circle used to annotate diminished chords: 0x00B0 ”DEGREE SIGN”
- plus sign used to annotate augmented chords: 0x002B ”PLUS SIGN”

Typedef Documentation

ARACircleOfFifthsIndex

```
typedef ARAInt32 ARACircleOfFifthsIndex
```

The root of a key signature or chord as an index (or angle) in the circle of fifths from 'C'. Enharmonic equivalents such as Db and C# are distinguished:

```
...
-5: Db
...
-1: F
0: C
1: G
2: D
...
7: C#
...
11: E#
...
```

Host Interfaces

Detailed Description

Audio Access Controller

Detailed Description

This interface allows plug-ins to read the audio data from the host in a random access order. It is used from multiple threads, both host and plug-in need to carefully observe the threading rules for each function. The basic design idea is that each audio reader is used single-threaded, but multiple audio readers can work concurrently (even on the same audio source). Audio readers can be considered random access iterators, and like most iterators operate on conceptually constant data structures. If the host changes sample rate, channel count or other audio source properties that the reader relies upon, the plug-in must discard any existing audio readers for the source and later re-create them based on the new configuration. The host can temporarily disable access to the audio source in order to control the exact timing of stopping the readers from accessing the source. Whenever an audio reader is destroyed, the plug-in is responsible for thread safety - it may need to block until a concurrent read operation on the I/O thread has finished. Hosts must take care in their audio reader implementation to avoid potential deadlocks in this situation. Note that when rendering, the audio source will often not be read in a consecutive order - depending on the edits the user applied at audio modification level, the access may jump back and forth quite often. The reader implementation should be optimized accordingly.

Host developer using C++ ARA Library can implement the ARA::Host::AudioAccessControllerInterface. For plug-in developers this interface is wrapped by the ARA::PlugIn::HostAudioAccessController.

Classes

- struct ARAAudioAccessControllerInterface

  Host interface: audio access controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed. More...

Typedefs

- typedef struct ARAAudioAccessControllerHostRefMarkupType * ARAAudioAccessControllerHostRef

  Reference to the host side representation of an audio access controller (opaque to the plug-in).
- typedef struct ARAAudioReaderHostRefMarkupType * ARAAudioReaderHostRef

  Reference to the host side representation of an audio reader (opaque to the plug-in).

Class Documentation

struct ARAAudioAccessControllerInterface

Host interface: audio access controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Synchronous reading: blocking until data is available

- ARAAudioReaderHostRef(* createAudioReaderForSource)(ARAAudioAccessControllerHostRef controllerHostRef, ARAAudioSourceHostRef audioSourceHostRef, ARABool use64BitSamples)

  Create audio reader instance to access sample data in an audio source. The format of the data is matching the format of the audio source, with a choice of reading samples as 32 bit or 64 bit (hosts must support both formats). Similar to the rules for creating content readers, the plug-in may call these functions only from calls in the "Audio Source Management" section of ARADocumentControllerInterface for the particular audio source the call is referring to, or from ARADocumentControllerInterface::endEditing() for any audio source. Contrary to content readers, audio readers are long-living objects that are not bound to a certain stack frame.
- ARABool(* readAudioSamples)(ARAAudioAccessControllerHostRef controllerHostRef, ARAAudioReaderHostRef audioReaderHostRef, ARASamplePosition samplePosition, ARASampleCount samplesPerChannel, void *const buffers[ ])

  Read audio samples. The samples are provided in non-interleaved buffers of double or float data, depending on whether use64BitSamples was set when creating the reader. The channel count equals the channel count of the audio source. The data alignment and byte order always matches the machine's native layout. If the requested sample range extends beyond the start or end of the audio source, the out-of-range samples should be filled with silence. This should not be treated as an error. This potentially blocking function may be called from any non-realtime thread (including threads for offline rendering), but not from more than one thread per reader at the same time. The host may decide to let the thread sleep until the requested data is available, the plug-in must be designed to deal with this without triggering priority inversion. The target buffer(s) are provided by the caller. Result is kARATrue upon success, or kARAFalse when there is a critical, nonrecoverable I/O error, such as a network failure while the file is being read from a server. In case of failing in this call, the buffers must be filled with silence and the host must notify the user about the problem in an appropriate way. The plug-in must deal gracefully with any such I/O errors, both during analysis and rendering.
- void(* destroyAudioReader)(ARAAudioAccessControllerHostRef controllerHostRef, ARAAudioReaderHostRef audioReaderHostRef)

  Destroy given audio reader created by the host. The caller must guarantee that the reader is currently not in use in some other thread. See ARAAudioAccessControllerInterface::createAudioReaderForSource() about the restrictions when to call this.

Archiving Controller

Detailed Description

This interface allows plug-ins to read and write archives with minimal memory impact. It also allows for displaying progress when archiving or unarchiving model graphs. Its functions may only be called during the archiving/unarchiving process.

Because of the potentially large size of the archives, ARA does not use simple monolithic memory blocks as known from many companion APIs. Instead, it establishes a stream-like archive format so that copying large blocks of memory can be avoided. Plug-ins that create large archives should use any mean of data reduction that is appropriate to reduce the archive size. For example, they may implement gzip compression. Since it has good knowledge of the characteristics of the data, it can configure the compression algorithms so that optimal results are achieved. Consequently, there's no point for host to try to compress the data any further with generic algorithms.

Hosts that support both 32 and 64 bit architectures shall be aware of the fact that ARA archive sizes are pointer-sized data types, so they will differ in bit width between these architectures. This must be taken into account when storing the archive size in the host's document structure. Also, when importing documents from 64 bit into 32 bit, the host must check whether the archive is small enough to be loaded at all (i.e. its size fits into 32 bits). If not, it shall refuse to load the archive and provide a proper error message. This may seem like a restriction, but the reasoning behind this is that if the archive already exceeds the available address space, the resulting unarchived graph will do so too.

There's no creation or destruction call for the archive readers/writers because they are provided by the host for the duration of the (un-)archiving process, so the lifetime is implicitly defined.

When using API generation 1 or older and loading an archive through the deprecated functions begin-/endRestoringDocumentFromArchive(), plug-ins may choose to access the associated archive reader upon either begin- or endRestoringDocumentFromArchive() or even upon both calls, as suitable for their implementation - hosts must be able to provide the requested data during the duration of both calls.

Host developer using C++ ARA Library can implement the ARA::Host::ArchivingControllerInterface. For plug-in developers this interface is wrapped by the ARA::PlugIn::HostArchivingController.

Classes

- struct ARAArchivingControllerInterface

  Host interface: archive controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed. *More...*

Typedefs

- typedef struct ARAArchivingControllerHostRefMarkupType * ARAArchivingControllerHostRef

  Reference to the host side representation of an archiving controller (opaque to the plug-in).
- typedef struct ARAArchiveReaderHostRefMarkupType * ARAArchiveReaderHostRef

  Reference to the host side representation of an archive reader (opaque to the plug-in).
- typedef struct ARAArchiveWriterHostRefMarkupType * ARAArchiveWriterHostRef

  Reference to the host side representation of an archive writer (opaque to the plug-in).

Class Documentation

struct ARAArchivingControllerInterface

Host interface: archive controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Reading Archives

- ARASize(∗ getArchiveSize)(ARAArchivingControllerHostRef controllerHostRef, ARAArchiveReaderHostRef archiveReaderHostRef)

  Query the size of the archive. This may only be called from ARADocumentControllerInterface::restoreObjectsFromArchive(), or if using API generation 1 from the deprecated begin-/endRestoringDocumentFromArchive() calls. Plug-ins must respect this size when reading the archive, reading beyond the end of the data is a programming error (and should thus be asserted by the host).
- ARABool(∗ readBytesFromArchive)(ARAArchivingControllerHostRef controllerHostRef, ARAArchiveReaderHostRef archiveReaderHostRef, ARASize position, ARASize length, ARAByte buffer[ ])

  Read bytes. This may only be called from ARADocumentControllerInterface::restoreObjectsFromArchive(), or if using API generation 1 from the deprecated begin-/endRestoringDocumentFromArchive() calls. Result is kARATrue upon success, or kARAFalse when there is a critical, nonrecoverable I/O error, such as a network failure while the file is being read from a server. In case of failing in this call, the host must notify the user about the problem in some appropriate way. The archive will not be restored by the plug-in then, it'll fall back into some proper initial state for the affected objects.
- ARAPersistentID(∗ getDocumentArchiveID)(ARAArchivingControllerHostRef controllerHostRef, ARAArchiveReaderHostRef archiveReaderHostRef)

  Query the document archive ID that the plug-in's factory provided when saving the archive. This may only be called from ARADocumentControllerInterface::restoreObjectsFromArchive(). Plug-ins can use this information to optimize their unarchiving code in case different archive formats are used depending on document archive ID. The returned pointer is owned by the host and must remain valid until the archive reader is destroyed. All hosts that support kARAAPIGeneration_2_0_Final or newer must implement this call.

Writing Archives

- ARABool(∗ writeBytesToArchive)(ARAArchivingControllerHostRef controllerHostRef, ARAArchiveWriterHostRef archiveWriterHostRef, ARASize position, ARASize length, const ARAByte buffer[ ])

  Write bytes. This may only be called from storeObjectsToArchive() or the deprecated storeDocumentToArchive(). Result is kARATrue upon success, or kARAFalse when there is a critical, nonrecoverable I/O error, such as a network failure while the file is being written to a server. In case of failing in this call, the host must notify the user about the problem in an appropriate way.
  Note that a plug-in should strive to write the data consecutively in a stream-like manner. Nevertheless, repositioning is needed to support chunk-style archives where the chunk length must be specified at the start of the chunk, but is not known until the chunk data has been fully created (and possibly compressed). In that case, the plug-in will need to "rewind" to the chunk size entry in the chunk header after writing the chunk and update it accordingly. As in most file APIs, any range of bytes that was skipped when writing should be filled with 0 by the host.

(Un-)Archiving Progress Information

- void(∗ notifyDocumentArchivingProgress)(ARAArchivingControllerHostRef controllerHostRef, float value)

  Message to the host signaling document save progress, value ranges from 0.0f to 1.0f. This may only be called from storeObjectsToArchive() or from the deprecated storeDocumentToArchive(). In order to keep CPU load low, plug-ins should try to keep the update rate for this call as low as about 1000 calls per archive, which equals increments of 0.1%.

- void(* notifyDocumentUnarchivingProgress)(ARAArchivingControllerHostRef controllerHostRef, float value)

  Message to the host signaling document save progress, value ranges from 0.0f to 1.0f. This may only be called from ARADocumentControllerInterface::restoreObjectsFromArchive(), or if using API generation 1 from the deprecated begin-/endRestoringDocumentFromArchive() and the associated model object creation/update calls guarded by them. In the deprecated form, the first call should be made from begin- and the last call from endRestoringDocumentFromArchive(). In order to keep CPU load low, plug-ins should try to keep the update rate for this call as low as about 1000 calls per archive, which equals increments of 0.1%.

Model Content Access Controller

Detailed Description

This optional interface provides access to host model data such as the musical context. Its functions may only be called from ARADocumentControllerInterface.create...() or update...() for the object currently created/updated, or from ARADocumentControllerInterface::endEditing() for any object.

Host developer using C++ ARA Library can implement the ARA::Host::ContentAccessControllerInterface. For plug-in developers this interface is wrapped by the ARA::PlugIn::HostContentAccessController.

Classes

- struct ARAContentAccessControllerInterface

  Host interface: content access controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed. More...

Typedefs

- typedef struct ARAContentAccessControllerHostRefMarkupType * ARAContentAccessControllerHostRef

  Reference to the host side representation of a content access controller (opaque to the plug-in).

Class Documentation

struct ARAContentAccessControllerInterface

Host interface: content access controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Content Reader Management

These functions mirror the content reader section in ARADocumentControllerInterface.

- ARABool(* isMusicalContextContentAvailable)(ARAContentAccessControllerHostRef controllerHostRef, ARAMusicalContextHostRef musicalContextHostRef, ARAContentType contentType)

  Query whether the given content type is currently available for the given musical context.

- ARAContentGrade(∗ getMusicalContextContentGrade)(ARAContentAccessControllerHostRef controllerHostRef, ARAMusicalContextHostRef musicalContextHostRef, ARAContentType contentType)

  Query the current quality of the information provided for the given musical context and content type.
- ARAContentReaderHostRef(∗ createMusicalContextContentReader)(ARAContentAccessControllerHostRef controllerHostRef, ARAMusicalContextHostRef musicalContextHostRef, ARAContentType contentType, const ARAContentTimeRange ∗range)

  Create a content reader for the given musical context and content type. This should only be called after availability has been confirmed using isMusicalContextContentAvailable(), and is mainly used to communicate the song timeline to the plug-in. The time range may be NULL, which means that the entire musical context shall be read. If a time range is specified, all events that at least partially intersect with the range will be read.
- ARABool(∗ isAudioSourceContentAvailable)(ARAContentAccessControllerHostRef controllerHostRef, ARAAudioSourceHostRef audioSourceHostRef, ARAContentType contentType)

  Query whether the given content type is currently available for the given audio source.
- ARAContentGrade(∗ getAudioSourceContentGrade)(ARAContentAccessControllerHostRef controllerHostRef, ARAAudioSourceHostRef audioSourceHostRef, ARAContentType contentType)

  Query the current quality of the information provided for the given audio source and content type.
- ARAContentReaderHostRef(∗ createAudioSourceContentReader)(ARAContentAccessControllerHostRef controllerHostRef, ARAAudioSourceHostRef audioSourceHostRef, ARAContentType contentType, const ARAContentTimeRange ∗range)

  Create a content reader for the given audio source and content type. This should only be called after availability has been confirmed using isAudioSourceContentAvailable(). The host may be able to provide meta-information such as a known tempo or a known set of notes used in the material, which may enable the plug-in to speed up or even skip completely certain analysis passes (often depending on the content grade of the information). The time range may be NULL, which means that the entire audio source shall be read. If a time range is specified, all events that at least partially intersect with the range will be read.
- ARAInt32(∗ getContentReaderEventCount)(ARAContentAccessControllerHostRef controllerHostRef, ARAContentReaderHostRef contentReaderHostRef)

  Query how many events the given reader exposes.
- const void ∗(∗ getContentReaderDataForEvent)(ARAContentAccessControllerHostRef controllerHostRef, ARAContentReaderHostRef contentReaderHostRef, ARAInt32 eventIndex)

  Query data of the given event of the given reader. The returned pointer is owned by the host and must remain valid until either getContentReaderDataForEvent() is called again or the content reader is destroyed.
- void(∗ destroyContentReader)(ARAContentAccessControllerHostRef controllerHostRef, ARAContentReaderHostRef contentReaderHostRef)

  Destroy the given content reader.

Model Update Controller Interface

Detailed Description

This optional host interface allows the host to be notified about content changes in the plug-in. Its functions may only be called from ARADocumentControllerInterface::notifyModelUpdates().

Host developer using C++ ARA Library can implement the ARA::Host::ModelUpdateControllerInterface. For plug-in developers this interface is wrapped by the ARA::PlugIn::HostModelUpdateController.

Classes

- struct ARAModelUpdateControllerInterface

  Host interface: model update controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed. More...

Typedefs

- typedef struct ARAModelUpdateControllerHostRefMarkupType * ARAModelUpdateControllerHostRef

  Reference to the host side representation of a model update controller (opaque to the plug-in).

Class Documentation

struct ARAModelUpdateControllerInterface

Host interface: model update controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.

- void(* notifyAudioSourceAnalysisProgress)(ARAModelUpdateControllerHostRef controllerHostRef, ARAAudioSourceHostRef audioSourceHostRef, ARAAnalysisProgressState state, float value)

  Message to the host signaling analysis progress, value ranges from 0.0f to 1.0f. The first message must be marked with kARAAnalysisProgressStarted, the last with kARAAnalysisProgressCompleted. This notification is intended solely for displaying a progress indication if desired, but not to trigger content reading for updating content information. That is instead done when receiving notifyAudioSourceContentChanged(), see below. Note that since the updates are polled by the host, any analysis may already have progressed somewhat by the time the start notification is actually delivered, so it may start with a progress larger than 0. It is even possible that an analysis fully completes before its start is notified, in which case the plug-in may choose not notify it at all. If the plug-in internally executes multiple analysis task per audio source simultaneously (for example because it splits them by content type), it must merge all internal progress into a single outer progress.

- void(* notifyAudioSourceContentChanged)(ARAModelUpdateControllerHostRef controllerHostRef, ARAAudioSourceHostRef audioSourceHostRef, const ARAContentTimeRange *range, ARAContentUpdateFlags flags)

  Message to the host when content of the given audio source changes. Not to be called if the change was explicitly triggered by the host, e.g. when restoring audio source state or if the host calls ARADocumentControllerInterface::updateAudioSourceContent(). When restoring state, plug-ins shall only make this call if the object state after loading is not equal to the archived state. This can happen e.g when unarchiving encounters errors, or when state is imported from older versions and converted to some updated model state. The time range may be NULL, this means that the entire audio source is affected. Note that this notification is what hosts will listen to when determining whether the state returned by isAudioSourceContentAnalysisIncomplete() has changed. In other words, the completion state of the analysis itself is considered to be a part of the overall content. Thus, if it changes, the overall content has changed, even if neither the grade nor the data for content readers did change (e.g. because an analysis failed to provide proper results).

- void(* notifyAudioModificationContentChanged)(ARAModelUpdateControllerHostRef controllerHostRef, ARAAudioModificationHostRef audioModificationHostRef, const ARAContentTimeRange *range, ARAContentUpdateFlags flags)

Message to the host when content of the given audio modification changes. Not to be called if the change was explicitly triggered by the host, e.g. when restoring audio modification state. When restoring state, plug-ins will only make this call if the object state after loading is not equal to the archived state. This can happen e.g when unarchiving encounters errors, or when state is imported from older versions and converted to some updated model state. The time range may be NULL, this means that the entire audio modification is affected. Whenever this notification is received, all playback regions for the given modification are affected as well if their range in modification time intersects with the given time range. Because of this, there was no separate change notification for playback region in ARA 1. This changed with the introduction of content-based fades in ARA 2.0, because playback region content can now change without any audio modification changes. For many use cases, hosts that implement ARA 2 do no longer concern themselves with this call (unless they are also hosting ARA 1 plug-ins and implement an according ARA 2 adapter).

- void(∗ notifyPlaybackRegionContentChanged)(ARAModelUpdateControllerHostRef controllerHostRef, ARAPlaybackRegionHostRef playbackRegionHostRef, const ARAContentTimeRange ∗range, ARAContentUpdateFlags flags)

  Message to the host when content of the given playback region changes (added in ARA 2.0). In ARA 1, region content updates were implicitly derived from audio modification updates, (see notifyAudioModificationContentChanged()), based on the assumption that the content of a playback region did depend only on the modification content and the transformation described in the playback region properties. Since ARA 2.0 however, plug-ins use the grouping provided by the region sequences to derive further parameters for the transformation, which enables them to perform content based fades or other custom adjustments. This means the content of any given region can change even if its modification and region properties remain unchanged, and that the host can no longer calculate the content of a region based on the content of its modification. Note that when kARAContentUpdateSignalScopeRemainsUnchanged is not set, this message also indicates that the host needs to update any cached copies of the rendered signal it may hold, and needs to query getPlaybackRegionHeadAndTailTime() for potential updates. The time range is specified in playback time and potentially covers head and tail time too. The time range may be NULL, this means that the entire playback region (including its head and tail time, which may be updated at the same time) is affected.

Enumeration Type Documentation

ARAAnalysisProgressState

```
enum ARAAnalysisProgressState : ARAInt32
```

Audio source analysis progress indication.

Enumerator

kARAAnalysisProgressStarted	Required as first state for any given analysis.
kARAAnalysisProgressUpdated	State for normal progress of an analysis.
kARAAnalysisProgressCompleted	Required as last state for any given analysis (no matter whether it completed or was cancelled).

Playback Controller Interface

Detailed Description

This optional host interface allows the plug-in to request playback state changes. The functions in this interface may be called concurrently, but not from render-threads. The host may choose to ignore any of these requests. The requests will typically be scheduled and executed with some delay. The current state of playback is transmitted via the companion API.

Host developer using C++ ARA Library can implement the ARA::Host::PlaybackControllerInterface. For plug-in developers this interface is wrapped by the ARA::PlugIn::HostPlaybackController.

Classes

- struct ARAPlaybackControllerInterface

  Host interface: playback controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed. More...

Typedefs

- typedef struct ARAPlaybackControllerHostRefMarkupType * ARAPlaybackControllerHostRef

  Reference to the host side representation of a playback controller (opaque to the plug-in).

Class Documentation

struct ARAPlaybackControllerInterface

Host interface: playback controller. As with all host interfaces, the function pointers in this struct must remain valid until all document controllers on the plug-in side that use it have been destroyed.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- void(* requestStartPlayback)(ARAPlaybackControllerHostRef controllerHostRef)

  Message to the host to start playback of our document.
- void(* requestStopPlayback)(ARAPlaybackControllerHostRef controllerHostRef)

  Message to the host to stop playback of our document.
- void(* requestSetPlaybackPosition)(ARAPlaybackControllerHostRef controllerHostRef, ARATimePosition timePosition)

  Message to the host to set the playback position of our document. Note that this may be called both when playing back or when stopped.
- void(* requestSetCycleRange)(ARAPlaybackControllerHostRef controllerHostRef, ARATimePosition startTime, ARATimeDuration duration)

  Message to the host to set the playback cycle range of our document.
- void(* requestEnableCycle)(ARAPlaybackControllerHostRef controllerHostRef, ARABool enable)

  Message to the host to enable or disable the playback cycle of our document.

Document Controller Instance

Detailed Description

The callbacks into the host are published by the host when creating a document controller on the plug-in side to maintain an ARA model graph. The instance struct and all interfaces and host refs therein must remain valid until the document controller is destroyed. The host can choose to create its controller objects per document controller instance, or it can share a single instance between all document controllers, whatever fits its needs. It may even mix-and-match both approaches per individual interface.

Classes

- struct ARADocumentControllerHostInstance

  The document controller host instance struct and all interfaces and refs therein must remain valid until all plug-in document controllers created with this struct have been destroyed by the host. More...

Class Documentation

struct ARADocumentControllerHostInstance

The document controller host instance struct and all interfaces and refs therein must remain valid until all plug-in document controllers created with this struct have been destroyed by the host.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

AudioAccessControllerInterface

Must always be supported.

- ARAAudioAccessControllerHostRef **audioAccessControllerHostRef**
- const ARAAudioAccessControllerInterface * **audioAccessControllerInterface**

ArchiveControllerInterface

Must always be supported.

- ARAArchivingControllerHostRef **archivingControllerHostRef**
- const ARAArchivingControllerInterface * **archivingControllerInterface**

ContentAccessControllerInterface

OPTIONAL INTERFACE: plug-in must check contentAccessControllerInterface is not NULL before calling!

- ARAContentAccessControllerHostRef **contentAccessControllerHostRef**
- const ARAContentAccessControllerInterface * **contentAccessControllerInterface**

ModelUpdateControllerInterface

OPTIONAL INTERFACE: plug-in must check modelUpdateControllerInterface is not NULL before calling!

- ARAModelUpdateControllerHostRef **modelUpdateControllerHostRef**
- const ARAModelUpdateControllerInterface * **modelUpdateControllerInterface**

PlaybackControllerInterface

OPTIONAL INTERFACE: plug-in must check playbackControllerInterface is not NULL before calling!

- ARAPlaybackControllerHostRef **playbackControllerHostRef**
- const ARAPlaybackControllerInterface * **playbackControllerInterface**

Plug-In Interfaces

Detailed Description

Partial Document Persistency

Detailed Description

These optional filters allow to only store a subset of the document graph into an archive, or only restore a subset of an archive into the document graph.

Classes

- struct ARARestoreObjectsFilter

  Optional filter when restoring objects.
  Allows the host to specify a subset of the persistent objects in the archive to restore in ARADocumentControllerInterface::restoreObjectsFromArchive().
  The given IDs refer to objects in the archive, but can optionally be mapped to those used in the current document. This may be necessary to resolve potential conflicts between persistent IDs from different documents when importing parts of one document into another (since persistent IDs are only required to be unique within a document, not across documents).
  The C++ ARA Library offers plug-in developers the ARA::PlugIn::RestoreObjectsFilter utility class to ease the implementation of partial persistency. More...
- struct ARAStoreObjectsFilter

  Optional filter when storing objects.
  Allows the host to specify a subset of the objects in the model graph to be stored in ARADocumentControllerInterface::storeObjectsToArchive().
  The C++ ARA Library offers plug-in developers the ARA::PlugIn::StoreObjectsFilter utility class to ease the implementation of partial persistency. More...

Class Documentation

struct ARARestoreObjectsFilter

Optional filter when restoring objects.

Allows the host to specify a subset of the persistent objects in the archive to restore in ARADocumentControllerInterface::restoreObjectsFromArchive().

The given IDs refer to objects in the archive, but can optionally be mapped to those used in the current document. This may be necessary to resolve potential conflicts between persistent IDs from different documents when importing parts of one document into another (since persistent IDs are only required to be unique within a document, not across documents).

The C++ ARA Library offers plug-in developers the ARA::PlugIn::RestoreObjectsFilter utility class to ease the implementation of partial persistency.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARABool documentData

  Flag to indicate whether the plug-in should include its private, opaque document state in the archive - see ARAStoreObjectsFilter::documentData for details.
- ARASize audioSourceIDsCount

  Length of audioSourceArchiveIDs and audioSourceCurrentIDs (if provided).
- const ARAPersistentID * audioSourceArchiveIDs

Variable-sized C array listing the persistent IDs of the archived audio sources to restore. The list may be empty, in which case count should be 0 and the pointer NULL.

- const ARAPersistentID * audioSourceCurrentIDs

  Optional variable-sized C array mapping each of the persistent IDs provided in audioSourceArchiveIDs to a potentially different persistent ID currently used for the audio sources to be restore in the current graph. If no mapping is desired, i.e. all audio source persistent IDs to restore match those in the current graph, the pointer should be NULL.
- ARASize audioModificationIDsCount

  Length of audioModificationArchiveIDs and audioModificationCurrentIDs (if provided).
- const ARAPersistentID * audioModificationArchiveIDs

  Variable-sized C array listing the persistent IDs of the archived audio modifications to restore. The list may be empty, in which case count should be 0 and the pointer NULL.
- const ARAPersistentID * audioModificationCurrentIDs

  Optional variable-sized C array mapping each of the persistent IDs provided in audioModificationArchiveIDs to a potentially different persistent ID currently used for the audio modifications to be restore in the current graph. If no mapping is desired, i.e. all audio modification persistent IDs to restore match those in the current graph, the pointer should be NULL.

struct ARAStoreObjectsFilter

Optional filter when storing objects.

Allows the host to specify a subset of the objects in the model graph to be stored in ARADocumentControllerInterface::storeObjectsToArchive().

The C++ ARA Library offers plug-in developers the ARA::PlugIn::StoreObjectsFilter utility class to ease the implementation of partial persistency.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARABool documentData

  Flag to indicate whether the plug-in should include its private, opaque document state in the archive. A typical example of private data is a fallback implementation in the plug-in for data not provided by the host. For example, if the host does not implement a chord track, a plug-in may need to implement this in its own UI in order to be fully usable. Since the host is not aware of this, the data must be stored privately at document level. This flag should be set to kARAFalse if the archive is intended for copy/paste or other means of data import/export between documents, or kARATrue if a host uses partial persistency as a general technique to store ARA documents for performance reasons (e.g. to avoid re-saving data that hasn't been changed, or to minimize sync activity when implementing collaborative editing across the network - note that such optimizations rely on proper update notifications by the plug-in). If implementing the latter, hosts will typically split the document into a partial archive that contains only the document data, plus a set of archives that each contain a single audio source. In such a setup, the audio modifications are either stored in the same archive as their underlying audio source, or each audio modification is separated into an archive of its own. The only restriction for such archive splicing (in addition to respecting the general data dependency rules for partial persistency outlined in the documentation of ARADocumentControllerInterface::restoreObjectsFromArchive()) is that when restoring, the partial archive which was saved with documentData == kARATrue is restored as last archive in the restore cycle, where the graph has its final structure and all object states are available.
- ARASize audioSourceRefsCount

  Length of audioSourceRefs.
- const ARAAudioSourceRef * audioSourceRefs

  Variable-sized C array listing the audio sources to store. The list may be empty, in which case count should be 0 and the pointer NULL.

- ARASize audioModificationRefsCount

  Length of audioModificationRefs.
- const ARAAudioModificationRef * audioModificationRefs

  Variable-sized C array listing the audio modifications to store. The list may be empty, in which case count should be 0 and the pointer NULL.

Processing Algorithm Selection

Detailed Description

Classes

- struct ARAProcessingAlgorithmProperties

  Processing algorithm description returned by ARADocumentControllerInterface::getProcessingAlgorithmProperties() Provides a unique identifier and a user-readable name of the algorithm, as displayed in the plug-in. The pointers contained in this struct must remain valid until the document controller that has provided the struct is destroyed. More...

Class Documentation

struct ARAProcessingAlgorithmProperties

Processing algorithm description returned by ARADocumentControllerInterface::getProcessingAlgorithmProperties() Provides a unique identifier and a user-readable name of the algorithm, as displayed in the plug-in. The pointers contained in this struct must remain valid until the document controller that has provided the struct is destroyed.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARAPersistentID persistentID

  ID for this particular processing algorithm.
- ARAUtf8String name

  Name as displayed by the plug-in (may be localized).

Document Controller

Detailed Description

ARA model objects are created and managed by the ARA Document Controller provided by the plug-in. The host uses the factory and management functions of the Document Controller to create a partial copy of its model translated into the ARA world. It is created using a factory which can be either retrieved by scanning the plug-in binaries for a dedicated factory or by requesting it from a living companion plug-in instance. The host is responsible for keeping the document controller alive as long as any objects created through it are still living (kind of implicit ref-counting). This means that its live time is independent of the companion plug-in instances - they may all be gone at some point but the ARA graph may still be accessed through its document controller. The host must also keep the document controller alive as long as any companion plug-in instance which it has bound to it is actively used. The actual destruction of the plug-in instance may be done later (to ease reference counting implementation), but rendering the plug-in, accessing its state or showing its UI is only valid as long as the ARA document controller it has been bound is still alive. Except for some rare, explicitly documented functions like

getPlaybackRegionHeadAndTailTime(), the document controller interface must always be called from the same thread - usually hosts will manage their internal model as well as the attached ARA graph from the application's main thread, triggered from the main run loop. If a host decides to use a different thread for maintaining the ARA model, it may need to implement some sort of locking so that its updates on the ARA model thread do not interfere concurrently with the main run loop's event processing as it drives the plug-in's UI code and notification system.

Plug-in developers using C++ ARA Library can implement the ARA::PlugIn::DocumentControllerInterface, or extend the already implemented ARA::PlugIn::DocumentController class as needed. For host developers this interface is wrapped by the ARA::Host::DocumentController.

Classes

- struct ARADocumentControllerInterface

  Plug-in interface: document controller. The function pointers in this struct must remain valid until the document controller is destroyed by the host. More...

Typedefs

- typedef struct ARADocumentControllerRefMarkupType * ARADocumentControllerRef

  Reference to the plug-in side representation of a document controller (opaque to the host).

Class Documentation

struct ARADocumentControllerInterface

Plug-in interface: document controller. The function pointers in this struct must remain valid until the document controller is destroyed by the host.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Destruction

- void(* destroyDocumentController)(ARADocumentControllerRef controllerRef)

  Destroy the controller and its associated document. The host must delete all objects associated with the document graph (audio sources, musical contexts etc.) before making this call. Note that the objects exported via the ARAPlugInExtensionInstance are not considered part of the document graph, their destruction may happen before or after destroying the document controller that they are bound to.

Link back to the factory

- const ARAFactory *(* getFactory)(ARADocumentControllerRef controllerRef)

  Query the static ARA factory that was used to create this controller. This provides a convenient traversal to the name of the plug-in, the description of its capabilities, its archive IDs etc.

Update management

- void(* beginEditing)(ARADocumentControllerRef controllerRef)

Start an editing session on a document. An editing session can contain an arbitrary set of modifications that belong together. Since many model edits can result in rather expensive updates on the plug-in side, this call allows for grouping the edits and postponing the updates until the new model state is final, which potentially saves some intermediate updates.

- void(∗ endEditing)(ARADocumentControllerRef controllerRef)

  End an editing session on a document. Note that when receiving this call, the plug-in will update any amount of internal state. These edits may lead to update notifications to the host, and the host may in turn read affected content from the plug-in and update its own model accordingly. One example for this the way that Melodyne maintains chords and scales associated with audio modifications. It copies this data from the musical context into the audio modifications, so that when editing regions the notes appear in the proper pitch grid. If moving playback regions in the song, these copies may need to be updated, and Melodyne will report the resulting audio modification content changes to the host. To ensure that any such follow-up updates are added to the same undo cycle, hosts that actively read plug-in content data should immediately (i.e. within the same undo frame) call notifyModelUpdates() after making this call.

- void(∗ notifyModelUpdates)(ARADocumentControllerRef controllerRef)

  Tell the plug-in to send all pending update notifications for the given document. This must be called periodically by the host whenever not editing nor restoring the document. Only when processing this call, the plug-in may call back into the host using ARAModelUpdateControllerInterface. Hosts must be aware after receiving beginEditing(), plug-ins may choose to postpone any subset of their internal state updates until the matching call to endEditing(). This means that if the host for some reason needs to wait for a specific update in the plug-in to occur (such as waiting for an analysis to finish) it must do so outside of pairs of beginEditing() and endEditing().

Document Persistency (extended in ARA 2.0)

- ARABool(∗ beginRestoringDocumentFromArchive)(ARADocumentControllerRef controllerRef, ARAArchiveReaderHostRef archiveReaderHostRef)

  Begin an unarchiving session of the document and its associated objects.
  Deprecated
  Since version 2_0_Final this call has been superseded by the combination of beginEditing() and restoreObjectsFromArchive(). This allows for optional filtering, but also simplifies both host and plug-in implementation.

- ARABool(∗ endRestoringDocumentFromArchive)(ARADocumentControllerRef controllerRef, ARAArchiveReaderHostRef archiveReaderHostRef)

  End an unarchiving session of the document and its associated objects.
  Deprecated
  Since version 2_0_Final this call has been superseded by the combination of endEditing() and restoreObjectsFromArchive(). This allows for optional filtering, but also simplifies both host and plug-in implementation.
  When using API generation 1 or older and using this call, the host must pass the same archiveReaderHostRef as used for beginRestoringDocumentFromArchive(). This way, plug-ins can choose to evaluate the archive upon beginRestoring() or endRestoring(), or even upon both calls if needed.

- ARABool(∗ storeDocumentToArchive)(ARADocumentControllerRef controllerRef, ARAArchiveWriterHostRef archiveWriterHostRef)

  Create an archive of the internal state of a given document and all its associated objects.
  Deprecated
  Since version 2_0_Final this call has been superseded by storeObjectsToArchive(), which allows for optional filtering, but is otherwise identical.

- ARABool(∗ restoreObjectsFromArchive)(ARADocumentControllerRef controllerRef, ARAArchiveReaderHostRef archiveReaderHostRef, const ARARestoreObjectsFilter ∗filter)

  Unarchive the internal state of the specified objects. This call can be used both for unarchiving entire documents and for importing arbitrary objects into an existing document. An unarchiving session is conceptually identical to an editing session: after starting the session, the host rebuilds the graph using the regular object creation calls, then makes this call to let the plug-in parse the archive and inject the archived internal state into the graph as indicated by the persistentIDs of the relevant objects. Similarly,

when importing objects, the host will perform an editing session and either create new objects or re-use existing objects (potentially adjusting their persistentID), then make this call to inject the imported state.
The optional filter allows for restoring only a subset of the archived states into the graph. It can be NULL, in which case all archived states with matching persistentIDs will be restored. In that case, the call sequence beginEditing(), restoreObjectsFromArchive(), endEditing() is equivalent to the deprecated begin-/endRestoringDocumentFromArchive() for ARA 1, which has been superseded by this call. The host is not required to restore all objects in the archive. Any archived states that are either filtered explicitly, or for which there is no object with a matching persistent ID in the current graph are simply ignored. Since persistent IDs are only required to be unique per document (and not globally), hosts may encounter persistent ID conflicts when importing data from other documents. The optional ARARestoreObjectsFilter provided for this call therefore allows to map between the IDs used in the archive and those used in the current graph if needed.
The host can make multiple calls to restoreObjectsFromArchive() within the same editing session to import objects from multiple archives in one operation. It may even decide to implement its persistency based on partial archives entirely, using several calls to storeObjectsToArchive() with varying filters to split the document into slices appropriate to its implementation, see ARAStoreObjectsFilter::documentData.
Result is kARAFalse if the access to the archive reader failed while trying to read the archive, or if decoding the data failed, kARATrue otherwise. Potential reason for failure include data corruption due to storage hardware failures, or broken dependencies when restoring partial archives as discussed above. The plug-in should try to recover as much state as possible in all cases, and the host should notify the user of such errors. If a failure happened already while reading the archive, the host is aware of this and can augment its error message to the user accordingly. If the failure happens inside the plug-in when decoding the data, the plug-in is responsible for guiding the user as good as possible, e.g. by listing or marking the affected objects. Note that since versioning is expressed through the ARA factory, the host must deal with potential versioning conflicts before making this call, and provide proper UI too.

- ARABool(∗ storeObjectsToArchive)(ARADocumentControllerRef controllerRef, ARAArchiveWriterHostRef archiveWriterHostRef, const ARAStoreObjectsFilter ∗filter)

  Create a partial archive of the internal state of the specified objects. Archives may only be created from documents that are not being currently edited. The optional filter allows for storing only a subset of the document graph into the archive. It can be NULL, in which case all objects in the graph will be stored. In that case, the call is equivalent to the deprecated storeDocumentToArchive(), which has been superseded by this call. Result is kARAFalse if the access to the archive writer failed while trying to write the archive, kARATrue otherwise. The host is responsible for alerting the user about archive write errors, see ARAArchivingControllerInterface::writeBytesToArchive(). Note that for creating ARA audio file chunk archives, storeAudioSourceToAudioFileChunk() must be used instead, so that the plug-in can pick the correct encoding and return the corresponding (compatible) document archive ID.

- ARABool(∗ storeAudioSourceToAudioFileChunk)(ARADocumentControllerRef controllerRef, ARAArchiveWriterHostRef archiveWriterHostRef, ARAAudioSourceRef audioSourceRef, ARAPersistentID ∗documentArchiveID, ARABool ∗openAutomatically)

  Create an archive of the internal state of the specified audio source suitable to be embedded into the underlying audio file as ARA audio file chunks, see ARA Audio File Chunks. Hosts must check ARAFactory::supportsStoringAudioFileChunks before enabling users to store audio file chunks for the given plug-in. Archives may only be created from documents that are not being currently edited.
  This call differs from using storeObjectsToArchive() with an ARAStoreObjectsFilter in that the plug-in may choose a different internal encoding more suitable for this use case, indicated by returning a `documentArchiveID` that is likely one of the ARAFactory::compatibleDocumentArchiveIDs rather than the ARAFactory::documentArchiveID. The plug-in also returns whether openAutomatically should be set in the audio file chunk. Result is kARAFalse if the access to the archive writer failed while trying to write the archive, kARATrue otherwise. The host is responsible for alerting the user about archive write errors, see ARAArchivingControllerInterface::writeBytesToArchive().

Document Management

As with all model graph edits, calling these functions must be guarded by beginEditing() and endEditing().

- void(∗ updateDocumentProperties)(ARADocumentControllerRef controllerRef, const ARADocumentProperties ∗properties)

Update the properties of the controller's document. All properties must be specified, the plug-in will determine which have actually changed.

Musical Context Management

As with all model graph edits, calling these functions must be guarded by beginEditing() and endEditing().

- ARAMusicalContextRef(∗ createMusicalContext)(ARADocumentControllerRef controllerRef, ARAMusicalContextHostRef hostRef, const ARAMusicalContextProperties ∗properties)

  Create a new musical context associated with the controller's document.
- void(∗ updateMusicalContextProperties)(ARADocumentControllerRef controllerRef, ARAMusicalContextRef musicalContextRef, const ARAMusicalContextProperties ∗properties)

  Update the properties of a given musical context. All properties must be specified, the plug-in will determine which have actually changed.
- void(∗ updateMusicalContextContent)(ARADocumentControllerRef controllerRef, ARAMusicalContextRef musicalContextRef, const ARAContentTimeRange ∗range, ARAContentUpdateFlags flags)

  Tell the plug-in to update the information obtainable via content readers for a given musical context. The time range may be NULL, this means that the entire musical content is affected. Creating a new musical context implies an initial content update for it - the host will call this explicitly only for later content updates.
- void(∗ destroyMusicalContext)(ARADocumentControllerRef controllerRef, ARAMusicalContextRef musicalContextRef)

  Destroy a given musical context. Destroying a musical context also implies removing it from its document. The musical context must no longer be referred to by any playback region when making this call.

Audio Source Management

As with all model graph edits, calling these functions must be guarded by beginEditing() and endEditing(), with the exception of enableAudioSourceSamplesAccess() (which is no model graph edit, but but still has to occur on the ARA model thread).

- ARAAudioSourceRef(∗ createAudioSource)(ARADocumentControllerRef controllerRef, ARAAudioSourceHostRef hostRef, const ARAAudioSourceProperties ∗properties)

  Create a new audio source associated with the controller's document. The newly created audio source has its sample data access initially disabled, an explicit call to enableAudioSourceSamplesAccess() is needed.
- void(∗ updateAudioSourceProperties)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, const ARAAudioSourceProperties ∗properties)

  Update the properties of a given audio source. Depending on which properties are changed (see documentation of ARAAudioSourceProperties), the host may not be able to make this call while the plug-in is reading data - in that case, use enableAudioSourceSamplesAccess() accordingly. All properties must be specified, the plug-in will determine which have actually changed.
- void(∗ updateAudioSourceContent)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, const ARAContentTimeRange ∗range, ARAContentUpdateFlags flags)

  Tell the plug-in that the sample data or content information for the given audio source has changed. Not to be called in response to ARAModelUpdateControllerInterface::notifyAudioSourceContentChanged(). The time range may be NULL, this means that the entire audio source is affected. When implementing this call, remember to also flush any caches of the sampled data if needed (see kARAContentUpdateSignalScopeRemainsUnchanged). Creating a new audio source always implies an initial content update for it, i.e. the host will call this function only for later content updates. Since audio sources are persistent, plug-ins should preferably postpone the initial content reading until endEditing() - if the host is restoring the audio source, this will remove the need to read initial content.

- void(∗ enableAudioSourceSamplesAccess)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARABool enable)

  Enable or disable access to a given audio source. This call allows the host to control the time when the plug-in may access sample data from the given audio source. Disabling access forces the plug-in to destroy all audio readers it currently has created for the affected audio source. This is a synchronous call, blocking until all currently executing reads of the audio source are finished. Access is disabled by default, hosts must explicitly enable it after creating an audio source. Since this call does not modify the model graph, it may be called outside the usual beginEditing() and endEditing() scope. Note that disabling access will also abort any analysis currently being executed for the audio source, making it necessary to start it from scratch when the access is enabled again. This means that enableAudioSourceSamplesAccess() is an expensive call that only should be made when necessary. It should not be (ab-)used to simply "pause ARA" whenever convenient.
- void(∗ deactivateAudioSourceForUndoHistory)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARABool deactivate)

  Deactivate the given audio source because it has become part of the undo history and is no longer used actively. The plug-in will cancel any pending analysis for this audio source and may free memory that is only needed when the audio source can be edited or rendered. Before deactivating an audio source, the host must deactivate all associated audio modifications, and the opposite order is required when re-activating upon redo. When deactivated, updating the properties or content of the audio source or reading its content is no longer valid. Like properties, deactivation is not necessarily persistent in the plug-in, so the host must call this explicitly when restoring deactivated audio sources. Note that with the introduction of partial persistency with ARA 2.0, hosts likely will prefer to simply create partial archives of deleted audio sources and manage these in their undo history rather than utilizing this call.
- void(∗ destroyAudioSource)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef)

  Destroy a given audio source. Destroying an audio source also implies removing it from its document. The host must delete all objects associated with the audio source (audio modifications etc.) before deleting the audio source. The host does not need to explicitly disable access to the audio source before making this call.

Audio Modification Management

- ARAAudioModificationRef(∗ createAudioModification)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARAAudioModificationHostRef hostRef, const ARAAudioModificationProperties ∗properties)

  Create a new audio modification associated with the given audio source.
- ARAAudioModificationRef(∗ cloneAudioModification)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef, ARAAudioModificationHostRef hostRef, const ARAAudioModificationProperties ∗properties)

  Create a new audio modification that copies the state of another given audio modification. The new modification will be associated with the same audio source. This call is used to create independent variations of the audio as opposed to creating aliases by merely adding playback regions to a given audio modification.
- void(∗ updateAudioModificationProperties)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef, const ARAAudioModificationProperties ∗properties)

  Update the properties of a given audio modification. All properties must be specified, the plug-in will determine which have actually changed.
- void(∗ deactivateAudioModificationForUndoHistory)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef, ARABool deactivate)

  Deactivate the given audio modification because it has become part of the undo history and is no longer used actively. The plug-in may free some memory that is only needed when the audio modification can be edited or rendered. Before deactivating an audio modification, the host must destroy all associated playback regions, and the opposite order is required when re-activating upon redo. When deactivated, updating the properties of the audio modification or reading its content is no longer valid. Like properties, deactivation

is not necessarily persistent in the plug-in, so the host must call this explicitly when restoring deactivated audio modifications. Note that with the introduction of partial persistency with ARA 2.0, hosts likely will prefer to simply create partial archives of deleted audio modifications and manage these in their undo history rather than utilizing this call.

- void(∗ destroyAudioModification)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef)

  Destroy a given audio modification. Destroying an audio modification also implies removing it from its audio source. The host must delete all objects associated with the audio modification (playback regions etc.) before deleting the audio modification.

- ARABool(∗ isAudioModificationPreservingAudioSourceSignal)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef)

  Some hosts such as Pro Tools provide indicators whether a given plug-in's current settings cause it to alter the sound of the original audio source, or preserve it so that bypassing/removing the plug-in would not change the perceived audible result (note that actual rendering involves using a playback region, which still may apply time-stretching or pitch-shifting to the audio modification's potentially unaltered output).
  Changes to this state are tracked via ARAModelUpdateControllerInterface::notifyAudioModificationContentChanged() with kARAContentUpdateSignalScopeRemainsUnchanged == false. Note that it is possible to perform other edits such as reassigning the chords associated with the audio modification which would not affect this state.
  It is valid for plug-in implementations to deliver false negatives here to reasonably limit the cost of maintaining the state. For example, if the plug-in does some threshold-based processing, but the signal happens to never actually reach the threshold, the plug-in still may report to alter the sound. Another example is pitch&time editing in a Melodyne-like plug-in: if notes are moved to a different pitch or time position so that this flag is cleared, but later the user manually moves them back to the original location, this might not cause this flag to turn back on. If however the user invokes undo, or some explicit reset command instead of the manual adjustment, then the plug-in should maintain this state properly.

Playback Region Management

As with all model graph edits, calling these functions must be guarded by beginEditing() and endEditing().

- ARAPlaybackRegionRef(∗ createPlaybackRegion)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef, ARAPlaybackRegionHostRef hostRef, const ARAPlaybackRegionProperties ∗properties)

  Create a new playback region associated with the given audio modification.

- void(∗ updatePlaybackRegionProperties)(ARADocumentControllerRef controllerRef, ARAPlaybackRegionRef playbackRegionRef, const ARAPlaybackRegionProperties ∗properties)

  Update the properties of a given playback region. All properties must be specified, the plug-in will determine which have actually changed.

- void(∗ destroyPlaybackRegion)(ARADocumentControllerRef controllerRef, ARAPlaybackRegionRef playbackRegionRef)

  Destroy a given playback region. Destroying a playback region also implies removing it from its audio modification. The playback region must no longer be referred to by any plug-in extension when making this call.

Content Reader Management

Content readers are not model objects but rather auxiliary objects to parse content without extensive data copying, following a standard iterator pattern. As is common with iterators, no changes may be made to the model graph while reading content. Prior to the final release of ARA 2.0, this meant that making any of the content related calls was limited to be done outside of pairs of beginEditing() and endEditing(). Since version 2_0_Final, it is also valid to use them while the document is in editing state, but no other call to this document controller may be made in-between a series of content related calls (except for

getFactory() and getPlaybackRegionHeadAndTailTime()). For example for a given audio source such a series of calls typically would be isAudioSourceContentAvailable(), getAudioSourceContentGrade(), createAudioSourceContentReader(), getContentReaderEventCount(), n times getContentReaderDataForEvent(), destroyContentReader(). Typically, content readers are temporary local objects on the stack of the ARA model thread, which naturally satisfies these conditions. This change is mainly intended to support partial persistency introduced with ARA 2.0, since it allows hosts to read content of imported audio sources/modifications to adjust the playback regions accordingly without toggling the editing state of the document back and forth.

- ARABool(∗ isAudioSourceContentAvailable)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARAContentType contentType)

  Query whether the given content type is currently available for the given audio source.
- ARABool(∗ isAudioSourceContentAnalysisIncomplete)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARAContentType contentType)

  Query whether an analysis of the given content type has been done for the given audio source. This call will typically be used when the host uses the plug-in as a detection engine in the background (i.e. without presenting the UI to the user). In that scenario, the host will trigger the analysis of the desired content types using requestAudioSourceContentAnalysis() and then wait until the plug-in calls ARAModelUpdateControllerInterface::notifyAudioSourceContentChanged(). From that call, the host will query the plug-in via isAudioSourceContentAnalysisIncomplete() to determine which of the analysis requests have completed. If the host did request a specific algorithm to be used, then the plug-in should return kARATrue here until the request was satisfied (or rejected).
- void(∗ requestAudioSourceContentAnalysis)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARASize contentTypesCount, const ARAContentType contentTypes[ ])

  Explicitly trigger a certain analysis. If the host wants to use the plug-in as detection engine, it needs to explicitly trigger the desired analysis, since otherwise the plug-in may postpone any analysis as suitable. To allow for optimizing the analysis on the plug-in side, all content types that are of interest for the host should be specified in a single call if possible.
  Note that the plug-in may choose to perform any additional analysis at any point in time if this is appropriate for its design. It will call ARAModelUpdateControllerInterface::notifyAudioSourceContentChanged() if such an analysis concludes successfully so that the host can update accordingly.
  The provided content types must be a non-empty subset of the plug-in's ARAFactory::analyzeableContentTypes. To request all analysis types exported by the plug-in, hosts can directly pass analyzeableContentTypes and -Count from the plug-in's ARAFactory. The contentTypes pointer may be only valid for the duration of the call, it must be evaluated inside the call, and the pointer must not be stored anywhere.
  Note that ARA 2.0 adds the capability for the host to also request the use of a certain processing algorithm via requestProcessingAlgorithmForAudioSource() - if both are used then the algorithm must be selected before requesting the analysis.
- ARAContentGrade(∗ getAudioSourceContentGrade)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARAContentType contentType)

  Query the current quality of the information provided for the given audio source and content type.
- ARAContentReaderRef(∗ createAudioSourceContentReader)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARAContentType contentType, const ARAContentTimeRange ∗range)

  Create a content reader for the given audio source and content type. This should only be called after availability has been confirmed using isAudioSourceContentAvailable(). The time range may be NULL, which means that the entire audio source shall be read. If a time range is specified, all events that at least partially intersect with the range will be read.
- ARABool(∗ isAudioModificationContentAvailable)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef, ARAContentType contentType)

Query whether the given content type is currently available for the given audio modification. Note that since ARA 2.0, reading at playback region level is recommended for most content types, see createAudioModificationContentReader().

- ARAContentGrade(∗ getAudioModificationContentGrade)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef, ARAContentType contentType)

  Query the current quality of the information provided for a given audio modification and content type. Note that since ARA 2.0, reading at playback region level is recommended for most content types, see createAudioModificationContentReader().

- ARAContentReaderRef(∗ createAudioModificationContentReader)(ARADocumentControllerRef controllerRef, ARAAudioModificationRef audioModificationRef, ARAContentType contentType, const ARAContentTimeRange ∗range)

  Create a content reader for the given audio modification and content type. This should only be called after availability has been confirmed using isAudioModificationContentAvailable(). The time range may be NULL, which means that the entire audio modification shall be read. If a time range is specified, all events that at least partially intersect with the range will be read. Note that with the introduction of region transitions in ARA 2.0, the content of a given playback region can no longer be externally calculated by the host based on the content of its underlying audio modification and the transformation flags. Instead, hosts should read such content directly at region level. This particularly applies to kARAContentTypeNotes - notes at borders will be adjusted on a per-region basis when using content based fades. Playback region content reading is available in ARA 1.0 already, thus such an implementation will be fully backwards compatible. Reading content at audio modification (or audio source) level is still valid and useful if the host needs access to the content in its original state, not transformed by a playback region, e.g. when implementing features such as tempo and signature detection through ARA.

- ARABool(∗ isPlaybackRegionContentAvailable)(ARADocumentControllerRef controllerRef, ARAPlaybackRegionRef playbackRegionRef, ARAContentType contentType)

  Query whether the given content type is currently available for the given playback region.

- ARAContentGrade(∗ getPlaybackRegionContentGrade)(ARADocumentControllerRef controllerRef, ARAPlaybackRegionRef playbackRegionRef, ARAContentType contentType)

  Query the current quality of the information provided for the given playback region and content type.

- ARAContentReaderRef(∗ createPlaybackRegionContentReader)(ARADocumentControllerRef controllerRef, ARAPlaybackRegionRef playbackRegionRef, ARAContentType contentType, const ARAContentTimeRange ∗range)

  Create a content reader for the given playback region and content type. This should only be called after availability has been confirmed using isPlaybackRegionContentAvailable(). The time range may be NULL, which means that the entire playback region shall be read, including its potential head and tail time. If a time range is specified, all events that at least partially intersect with the range will be read. The time range must be given in playback time, and the time stamps provided by the content reader are in playback time as well.

- ARAInt32(∗ getContentReaderEventCount)(ARADocumentControllerRef controllerRef, ARAContentReaderRef contentReaderRef)

  Query how many events the given reader exposes.

- const void ∗(∗ getContentReaderDataForEvent)(ARADocumentControllerRef controllerRef, ARAContentReaderRef contentReaderRef, ARAInt32 eventIndex)

  Query data of the given event of the given reader. The returned pointer is owned by the plug-in and must remain valid until either getContentReaderDataForEvent() is called again or the content reader is destroyed.

- void(∗ destroyContentReader)(ARADocumentControllerRef controllerRef, ARAContentReaderRef contentReaderRef)

  Destroy the given content reader.

Region Sequence Management (added in ARA 2.0)

As with all model graph edits, calling these functions must be guarded by beginEditing() and endEditing().

- ARARegionSequenceRef(∗ createRegionSequence)(ARADocumentControllerRef controllerRef, ARARegionSequenceHostRef hostRef, const ARARegionSequenceProperties ∗properties)

  Create a new region sequence associated with the controller's document.
- void(∗ updateRegionSequenceProperties)(ARADocumentControllerRef controllerRef, ARARegionSequenceRef regionSequenceRef, const ARARegionSequenceProperties ∗properties)

  Update the properties of a given region sequence. All properties must be specified, the plug-in will determine which have actually changed.
- void(∗ destroyRegionSequence)(ARADocumentControllerRef controllerRef, ARARegionSequenceRef regionSequenceRef)

  Destroy a given region sequence. The region sequence must no longer be referred to by any playback region when making this call.

Playback Region Head and Tail Time (added in ARA 2.0)

Depending on the specific DSP performed by the plug-in, the resulting signal for any region may exceed the region borders. Among other use cases such as delays, this will typically happen when the content based fades are enabled. Reporting this potential excess of signal both at start and end of a region allows for the host to adjust their rendering so that it includes head and tail time, or else fade the signal at the region borders to avoid crackles. This function can be called either from the model thread or from any (realtime or offline) audio rendering thread. If edits that the host performs on the model affect the head or tail times (such as toggling content based fades on or off), the new values can only be reliably queried on the model thread once endEditing() has returned. On audio threads, they can only be reliably queried until all render calls that may have occurred concurrently with endEditing() have returned.

- void(∗ getPlaybackRegionHeadAndTailTime)(ARADocumentControllerRef controllerRef, ARAPlaybackRegionRef playbackRegionRef, ARATimeDuration ∗headTime, ARATimeDuration ∗tailTime)

  Query the current head and tail time of a given playback region. Note that when a plug-in optimizes region transitions, the head and tail of any given region can change upon any model edit, even if it is not directly affected by the edit. Also, in order to properly track interaction between regions, plug-ins may lazily update this information upon endEditing(). Plug-ins will call notifyPlaybackRegionContentChanged() whenever these values change. headTime and tailTime must not be NULL. Host may query this often, so plug-ins should cache the value if there's any expensive calculation involved. Note that most companion APIs also feature a tail time concept. For playback renderer plug-in instances, the tail time reported via the companion API should be equal to or greater than the maximum of the tail times of all playback regions currently associated with the given renderer (i.e. the tail for any given playback region may be somewhat shorter then the companion API tail, depending on the region's content).

Processing Algorithm Selection (added in ARA 2.0)

Many plug-ins feature a set of different processing algorithms, each suited for a given kind of audio material. When analyzing the audio, plug-ins either try to automatically determine the appropriate algorithm that best matches the nature of the given material, or delegate this decision to the user. This fundamental selection may affect all further processing of the audio source - in addition to the analysis results, internal editing and rendering parameters and even the entire UI may change. Accordingly, changing this selection may invalidate some or all edits done by the user in the audio modifications based on the audio source. To avoid such data loss, making this selection appropriately should always be the first step when working with audio sources. By exporting the list of processing algorithms here, hosts can implement this selection, based on their knowledge about the origin of the audio material. Some hosts allow for configuring their built-in detection in a very similar way on a per-track basis, usually via a menu. They can use this technique for compatible ARA plug-ins as well. Host might also be able to implicitly deduce an appropriate processing algorithm. For example, when adding a new audio source to a region sequence that already refers to a set of previously analyzed audio sources which all use the same detection algorithm, this algorithm is likely appropriate for the new audio source as well. Along the same lines,

when recording a new take of some material that already has been analyzed by the plug-in, it is reasonable to use the same algorithm for the new take. Note that each processing algorithm must implement detection for all content types exported by the plug-in, albeit the detection quality may likely be different for some types. Plug-ins should keep the list of available processing algorithms restricted to the smallest reasonable set - after all, in an ideal (but currently not achievable) implementation, a single detection algorithm would be powerful enough to cover all types of material flawlessly.

- ARAInt32(∗ getProcessingAlgorithmsCount)(ARADocumentControllerRef controllerRef)

  Return the count of processing algorithms provided by the plug-in. If this optional method is not implemented or the call returns 0, then the plug-in does not support algorithm selection through the host and the other related functions below must not be called.

- const ARAProcessingAlgorithmProperties ∗(∗ getProcessingAlgorithmProperties)(ARADocumentControllerRef controllerRef, ARAInt32 algorithmIndex)

  List of processing algorithms provided by the plug-in, described by their properties. This method must be implemented if getProcessingAlgorithmsCount() is implemented. Provides a unique identifier and a user-readable name of the algorithm as displayed in the plug-in. The host should present the algorithms to the user in the order of this list, e.g. in a menu. For a given version of the plug-in, the count and the order and values of the persistentIDs must be the same, while the names may depend on localization settings that can be different on different machines or between individual runs of the host. The list may however change between different versions of the plug-in. Both hosts and plug-ins must implement fallbacks for loading a document that contains an processing algorithm persistentID which is no longer supported by the plug-in. The pointers returned by this calls must remain valid until the document controller is destroyed.

- ARAInt32(∗ getProcessingAlgorithmForAudioSource)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef)

  Query currently used processing algorithm for a given audio source. This method must be implemented if getProcessingAlgorithmsCount() is implemented. After the plug-in has concluded an analysis (as indicated via ARAModelUpdateControllerInterface::notifyAudioSourceContentChanged()), the host can query which processing algorithm was used and update its UI accordingly. This is particularly relevant if the host did explicitly request an analysis with a specific algorithm, but the plug-in was unable to satisfy this request for some reason. Similarly, the user may have changed the algorithm through the plug-in's UI. Note that until the first analysis has completed (i.e. as long as isAudioSourceContentAnalysisIncomplete() returns kARATrue), the value returned here may be an abstract default, not related to the actual audio source content. This will e.g. typically be the case in Melodyne, which uses an "automatic mode" as default until the first analysis has determined the actual processing algorithm that is suitable for the material. The call should not be made while isAudioSourceContentAnalysisIncomplete() returns kARATrue, because the returned value is likely stale.

- void(∗ requestProcessingAlgorithmForAudioSource)(ARADocumentControllerRef controllerRef, ARAAudioSourceRef audioSourceRef, ARAInt32 algorithmIndex)

  Request that any future analysis of the given audio source should use the given processing algorithm. This method must be implemented if getProcessingAlgorithmsCount() is implemented. This both affects any analysis requested by the host via requestAudioSourceContentAnalysis() as well as any analysis done by the plug-in on demand. Since this typically results in a model graph edit, calling this functions must be guarded by beginEditing() and endEditing(). Note that the plug-in is not required to heed this request if its internal state suggest otherwise, or if the user switches actively to a different algorithm. Also, some algorithms may be "meta" algorithms that will be replaced by a different actual algorithm, such as the "automatic" default algorithm in Melodyne which will pick an appropriate algorithm from the remaining list of algorithms when doing the initial analysis.

License Management (added in ARA 2.0)

If using ARA plug-ins for regular editing, their UI is shown and they can implement all license handling fully transparent for the host within their regular UI. If however the host is using the plug-in as internal engine for analysis (e.g. audio-to-MIDI conversion) or for time-stretching, it will typically not open the plug-in UI, and accordingly must then deal with potentially missing licenses that the plug-in requires to perform the desired tasks. As an example, Melodyne's analysis capabilities are only available when running Melodyne essential or higher, i.e. they are not supported by an unlicensed "playback-only" installation.

- ARABool(∗ isLicensedForCapabilities)(ARADocumentControllerRef controllerRef, ARABool runModalActivationDialogIfNeeded, ARASize contentTypesCount, const ARAContentType contentTypes[ ], ARAPlaybackTransformationFlags transformationFlags)

  With this optional call, hosts can test whether the current license state of the plug-in allows for requesting analysis of the given content types and rendering the given playback transformations (see requestAudioSourceContentAnalysis() and ARAPlaybackRegionProperties::transformationFlags). The host can also optionally instruct the plug-in to run a modal licensing dialog if the current license is not sufficient to perform the selected engine tasks, so that the user can review and adjust the licensing accordingly, such downloading a license from their respective user account or even purchase an upgrade that enables the requested features.
  The provided content types must be a subset of the plug-in's ARAFactory::analyzeableContentTypes. To request all analysis types exported by the plug-in, hosts can directly pass analyzeableContentTypes and -Count from the ARAFactory. The contentTypes pointer may be only valid for the duration of the call, it must be evaluated inside the call, and the pointer must not be stored anywhere. If not intending to use analysis, the count should be 0 and the array pointer NULL. The transformationFlags must be a subset of the plug-in's ARAFactory::supportedPlaybackTransformationFlags, and may be kARAPlaybackTransformationNoChanges if not intending to use transformations. The call returns kARATrue if the (potentially updated) license is sufficient to perform the requested tasks.

Document Controller Instance

Detailed Description

The callbacks into the plug-in are published by the plug-in when the host requests the creation of a document controller via the factory. The instance struct and all interfaces and host refs therein must remain valid until the document controller is destroyed. The plug-in can choose to create its controller objects per document controller instance, or it can share a single instance between all document controllers, whatever fits its needs. It may even mix-and-match both approaches per individual interface.

Classes

- struct ARADocumentControllerInstance

  The document controller instance struct and all interfaces and refs therein must remain valid until the document controller is destroyed by the host. More...

Class Documentation

struct ARADocumentControllerInstance

The document controller instance struct and all interfaces and refs therein must remain valid until the document controller is destroyed by the host.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Document Controller Interface

- ARADocumentControllerRef **documentControllerRef**
- const ARADocumentControllerInterface ∗ **documentControllerInterface**

Plug-In Factory

Detailed Description

Static entry into ARA, allows to create ARA objects.

Classes

- struct ARAInterfaceConfiguration

  API configuration. This configuration struct allows for setting the desired API version, the debug callback etc. Note that a pointer to this struct is only valid for the duration of the call to initializeARAWithConfiguration() - the data must be fully evaluated/copied inside the call. More...
- struct ARAFactory

  Static plug-in factory. All pointers herein must remain valid as long as the binary is loaded. The declaration of this struct will not change when updating to a new generation of the API, only additions are possible. More...

Class Documentation

struct ARAInterfaceConfiguration

API configuration. This configuration struct allows for setting the desired API version, the debug callback etc. Note that a pointer to this struct is only valid for the duration of the call to initializeARAWithConfiguration() - the data must be fully evaluated/copied inside the call.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARAAPIGeneration desiredApiGeneration

  Defines the API generation to use, must be within the range of supported generations.
- ARAAssertFunction * assertFunctionAddress

  Pointer to the global assert function address. Be aware that this is a pointer to a function pointer, not the function pointer itself! This indirection is necessary so that plug-ins can inject their debug code if needed. The pointer must be the same for all instances that use the same API generation. It must be always provided by the host, but can point to NULL to suppress debugging in release versions. It must remain valid until uninitializeARA() is called.

struct ARAFactory

Static plug-in factory. All pointers herein must remain valid as long as the binary is loaded. The declaration of this struct will not change when updating to a new generation of the API, only additions are possible.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Factory and Global Initialization

The following data members and functions are involved when loading/unloading the plug-in binary.

All function calls in this header may only be made between a single call to initializeARAWithConfiguration() and another single call to uninitializeARA(). Typically, initialization/uninitialization is done right after loading/right before unloading the binary, but it is valid to call them multiple times without loading/unloading the binary in-between, provided the described order of calls is respected.

- ARAAPIGeneration lowestSupportedApiGeneration

  Lower bound of supported ARAAPIGeneration.
- ARAAPIGeneration highestSupportedApiGeneration

  Upper bound of supported ARAAPIGeneration.
- ARAPersistentID factoryID

  Unique and versioned plug-in identifier. This ID must be globally unique and identifies the plug-in's document controller class created by this factory at runtime. The ID includes versioning, it must be updated if e.g. the plug-in's (compatible) document archive ID(s) or its analysis or playback transformation capabilities change. Host applications can therefore use it to trigger cache updates if they implement a plug-in caching mechanism to avoid scanning all plug-ins each time the program is launched. If a given plug-in supports multiple companion APIs, it will return the same ID across all companion APIs, allowing the host to choose which API to use for this particular plug-in. See Managing ARA Archives for more information.
- void(∗ initializeARAWithConfiguration)(const ARAInterfaceConfiguration ∗config)

  Start up ARA with the given configuration.
- void(∗ uninitializeARA)(void)

  Shut down ARA.

User-Presentable Meta Information

This information is only intended for display purposes and should never be used for flow control. In addition to all other potential uses, this information should always be stored alongside any document archive that the host creates in order to be able to provide a proper error message if the archive is being restored on a different system where the plug-in is not installed (or only an older version of it). See Managing ARA Archives for more information.

- ARAUtf8String plugInName

  Name of the plug-in to display to the user.
- ARAUtf8String manufacturerName

  Name of the manufacturer of the plug-in to display to the user.
- ARAUtf8String informationURL

  Web page to refer the user to if they need further information about the plug-in.
- ARAUtf8String version

  Version string of the plug-in to display to the user.

Document Controller and Document Archives

This section of the struct deals with creating document controllers and handling versioned document archiving based on archive IDs. Note that in addition to the actual document archive and its ID, the host should save the above meta information about the plug-in alongside so it can do proper error handling, see above.

- const ARADocumentControllerInstance ∗(∗ createDocumentControllerWithDocument)(const ARADocumentControllerHostInstance ∗hostInstance, const ARADocumentProperties ∗properties)

Factory function for the document controller. This call is used both when creating a new document from scratch and when restoring a document from an archive.

- ARAPersistentID documentArchiveID

  Identifier for document archives created by the document controller. This ID must be globally unique and is shared only amongst document controllers that create the same archives and produce the same render results based upon the same input data. This means that the ID must be updated if the archive format changes in any way that is no longer downwards compatible. See Managing ARA Archives for more information.

- ARASize compatibleDocumentArchiveIDsCount

  Length of compatibleDocumentArchiveIDs.

- const ARAPersistentID * compatibleDocumentArchiveIDs

  Variable-sized C array listing other identifiers of archives that the document controller can import. This list is used for example when updating the data structures inside the controller. It is ordered descending, expressing a preference which state to load in case a document contains more than one compatible archive of older versions. The list may be empty, in which case count should be 0 and the pointer NULL.

Capabilities

If the host wants to utilize the plug-in as an internal engine for analysis or time-stretching, it can use this information to determine if the plug-in is actually capable of handling its needs. Further, the host may want to restrict the user from doing operations that make no sense for the given plug-in, such as trying to time-stretch with a plug-in that cannot perform this task. Note that even if a plug-in cannot analyze a given content type, it still may provide a UI where the user can edit that content type, so it may support content reading for types missing here.

- ARASize analyzeableContentTypesCount

  Length of analyzeableContentTypes.

- const ARAContentType * analyzeableContentTypes

  Variable-sized C array listing the content types for which the plug-in can perform an analysis. This list allows the host to determine whether the plug-in can be used as analysis engine in the background (i.e. without presenting the UI to the user) to achieve certain host features. Note that the plug-in may support more content types than listed here for some or all objects in the graph, but may rely on user interaction when dealing with those content types. The list may be empty, in which case count should be 0 and the pointer NULL.

- ARAPlaybackTransformationFlags supportedPlaybackTransformationFlags

  Set of transformations that the plug-in supports when configuring playback regions. These flags allow the host to determine whether e.g. the plug-in can be used as time-stretch engine.

- ARABool supportsStoringAudioFileChunks

  Flag whether the plug-in supports exporting ARA audio file chunks via ARADocumentControllerInterface::storeAudioSourceToAudioFileChunk(). Note that reading such chunks is unaffected by this flag - as long as the documentArchiveID in the chunk is compatible, the plug-in must be able to read the data via ARADocumentControllerInterface::restoreObjectsFromArchive().

Plug-In Extension

Detailed Description

The plug-in extension provides ARA-specific additional functionality per companion plug-in instance. On a conceptual level, the plug-in extension is not an object of its own, but merely a set of additional interfaces of the companion plug-in instance, augmenting it with a few ARA-specific features in a fashion that is independent from the actual companion API in use. Accordingly, it is coupled 1:1 to the companion plug-in instance and its lifetime matches the lifetime of the companion plug-in instance (no separate destruction function needed). Along the same lines, plug-in extensions themselves are not persistent.

The plug-in extension is exposed towards the host when it binds a plug-in instance as created by the

companion APIs to a specific ARA model graph, represented by its associated document controller. This setup call is executed via a vendor-specific extension of the companion API and may only be made once. It shifts the ”normal” companion plug-in into the ARA world, and once established, this coupling cannot be undone, it remains active until the plug-in instance is destroyed.

Note that both performing the explicit binding and the implicit unbinding upon destruction will likely need to access plug-in internal data structures shared with with the document controller implementation. To avoid adding costly thread safety measures when maintaining this shared state, hosts should always perform these operations from the document controller thread (typically the main thread). This restriction may or may not apply when using the same companion API without ARA, so host developers might need to add extra precaution for the ARA case.

When ARA is enabled, the renderer behavior has slightly different semantics compared to the non-ARA use case. Since ARA renderers are essentially generators that use non-realtime data to generate realtime signals, they do not use the realtime input signal for processing. Playback renderers will simply ignore their inputs, but editor renderers will always add their output signal to the input signal provided by the host. If a plug-in assumes both rendering roles, playback rendering will already ignore the inputs, so the editor rendering will directly add to the playback output, not to the input.

Since ARA 2.0, the host can explicitly establish the roles that the given instance will assume in its specific implementation upon binding the plug-in instance to the ARA document controller. Each role is associated with a dedicated feature set that only is available when the particular role has been established. Depending on the chosen roles, the following calls control which playback regions are to be rendered according to which rule. Separating roles allows for more flexible ARA integrations and optimizes resource usage. A host could for example use a playback renderer plug-in instance playback region, plus one plug-in instance per track for editor rendering and viewing all regions on that track. Amongst other behavior, the roles heavily affect the relationship between plug-in instances and playback regions. For rendering, each plug-in extension can handle multiple playback regions if desired, albeit the semantics for modifying the set of associated regions per renderer are somewhat different between playback and editor renderers, see below. For editor view purposes, the relationship is not explicit to accommodate for a very broad range of user interface concepts that need to interact with the API. Generally, each editor view is associated with all playback regions in the document controller to which the plug-in is bound. However, typically only a varying subset of those regions will be shown at any point in time, depending on the intrinsic feature set of the plug-in, and reflecting the selection that the user has performed in the host - see notifySelection().

Classes

- struct ARAPlugInExtensionInstance

  The plug-in extension instance struct and all interfaces and refs therein must remain valid until the companion plug-in is destroyed by the host. Note that the companion plug-in destruction may happen before or after destroying the document controller it has been bound to, plug-ins must handle both possible destruction orders. Plug-ins must provide all interfaces that have been requested by the host through the role assignment, and suppress interfaces explicitly excluded by the roles - e.g. if the host did not assign kARAEditorRendererRole even it was known, editorRendererInterface will be NULL. More...

Class Documentation

struct ARAPlugInExtensionInstance

The plug-in extension instance struct and all interfaces and refs therein must remain valid until the companion plug-in is destroyed by the host. Note that the companion plug-in destruction may happen before or after destroying the document controller it has been bound to, plug-ins must handle both possible destruction orders. Plug-ins must provide all interfaces that have been requested by the host through the role assignment, and suppress interfaces explicitly excluded by the roles - e.g. if the host did not assign kARAEditorRendererRole even it was known, editorRendererInterface will be NULL.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**
- ARAPlugInExtensionRef **plugInExtensionRef**
- const ARAPlugInExtensionInterface * **plugInExtensionInterface**

ARA2 Instance Roles

- ARAPlaybackRendererRef **playbackRendererRef**
- const ARAPlaybackRendererInterface * **playbackRendererInterface**
- ARAEditorRendererRef **editorRendererRef**
- const ARAEditorRendererInterface * **editorRendererInterface**
- ARAEditorViewRef **editorViewRef**
- const ARAEditorViewInterface * **editorViewInterface**

Enumeration Type Documentation

ARAPlugInInstanceRoleFlags

`enum ARAPlugInInstanceRoleFlags : ARAInt32`
Plug-in instance role flags.

Enumerator

kARAPlaybackRendererRole	Role: playback render. Plug-in instances fulfilling this role are performing playback rendering, both for realtime song playback or for offline or realtime bounces/exports. Playback render plug-ins can be responsible for more than one playback region at a time, although hosts may prefer to use a 1:1 relationship to simplify their implementation and easily allow for further per-region processing in the host (fades, region gain). When dealing with looped regions and using the content based fades, a trivial optimization of the above pattern is to use the same renderer instance for all consecutive repetitions of the loop, which still allows for properly applying region gain and fades at the start or end of the consecutive repetitions. Another viable approach would be to use as many playback renderer instances per track as there are concurrently sounding regions on the track (i.e. including head and tail time), and distribute the regions in a round-robin fashion across those renderers so that in each renderer the regions never overlap. This way, the host could extract a separate signal per region with a minimal count of playback renderers. Note that there may be several playback renderers per playback region, for example if a host executes an export as background tasks that run concurrently with realtime playback. A playback render plug-in will replace its inputs with the rendered signal. If it does not also have editor rendering responsibilities, it does not need to be rendered while stopped, and during playback only needs to be rendered for the range covered by the region, plus its head and tail time. Playback renderers are transient, the host does not need to store the state of these instances via the companion API (unless they also fulfill kARAEditorViewRole or other persistent roles).

Enumerator

kARAEditorRendererRole	Role: editor render. Plug-in instances fulfilling this role are performing auxiliary realtime rendering that is only used to support the editing process, such as metronome clicks or playing the pitch of a note while it's being dragged to a different pitch. There should only be one plug-in instance that handles editor rendering for any given playback region. Since it is up to the plug-in to decide when it will generate previewing output based on the editing, it needs to be permanently rendered by the host. If a editor rendering plug-in is not responsible for playback, it will always forward its input signal to the output, adding its preview signal as needed. Otherwise it will add the signal to the playback render output described above. Editor renderers are transient, the host does not need to store the state of these instances via the companion API (unless they also fulfill kARAEditorViewRole or other persistent roles).
kARAEditorViewRole	Role: editor view. Plug-in instances fulfilling this role can be used to display a GUI. Unlike rendering for playback or editing, the view role is not tied to individual playback regions or region sequences, but rather to all regions and sequences within the document controller to which the given plug-in instance is bound. Selection and hiding of host entities is communicated to editor view plug-ins so that they can dynamically show a proper subset of the full document graph. Editor view plug-ins may contain some state related to user interface configuration, and thus their state needs to be persistent via the companion API.

Playback Renderer Interface (Added In ARA 2.0)

Detailed Description

See kARAPlaybackRendererRole.

Plug-in developers using C++ ARA Library can implement the ARA::PlugIn::PlaybackRendererInterface, or extend the already implemented ARA::PlugIn::PlaybackRenderer class as needed. For host developers this interface is wrapped by the ARA::Host::PlaybackRenderer.

Classes

- struct ARAPlaybackRendererInterface

  Plug-in interface: playback renderer. The function pointers in this struct must remain valid until the companion API plug-in instance (and accordingly its plug-in extension) is destroyed by the host. More...

Typedefs

- typedef struct ARAPlaybackRendererRefMarkupType * ARAPlaybackRendererRef

  Reference to the plug-in side representation of a playback renderer (opaque to the host).

Class Documentation

struct ARAPlaybackRendererInterface

Plug-in interface: playback renderer. The function pointers in this struct must remain valid until the companion API plug-in instance (and accordingly its plug-in extension) is destroyed by the host.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Assigning the playback region(s) for playback rendering

Specify the playback region(s) for which the this plug-in instance provides playback rendering. (and, if using ARA1 which does not distinguish roles: also editor rendering and view). These calls can only be made when the plug-in is not in render-state (”not active” in VST3 speak, ”not initialized” in AU-speak), however the host may display the UI for the plug-in while making these calls. They can be made both inside or outside any editing or restoration cycles of the associated ARA document controller, but doing them between ARADocumentControllerInterface::beginEditing() and ARADocumentControllerInterface::endEditing() when feasible may yield better performance since all updates can be calculated together. When called outside beginEditing() and endEditing(), the changes may not be audible until the next render call that starts after the plug-in has returned from the respective call. The calls shall be made on the same thread that also hosts the document controller. The host is not required to remove the playback regions before destroying the plug-in. If multiple regions are set, they will all be sounding concurrently in case they overlap.

- void(∗ **addPlaybackRegion**)(ARAPlaybackRendererRef playbackRendererRef, ARAPlaybackRegionRef playbackRegionRef)
- void(∗ **removePlaybackRegion**)(ARAPlaybackRendererRef playbackRendererRef, ARAPlaybackRegionRef playbackRegionRef)

Editor Renderer Interface (Added In ARA 2.0)

Detailed Description

See kARAEditorRendererRole.

Plug-in developers using C++ ARA Library can implement the ARA::PlugIn::EditorRendererInterface, or extend the already implemented ARA::PlugIn::EditorRenderer class as needed. For host developers this interface is wrapped by the ARA::Host::EditorRenderer.

Classes

- struct ARAEditorRendererInterface

  Plug-in interface: editor renderer. The function pointers in this struct must remain valid until the companion API plug-in instance (and accordingly its plug-in extension) is destroyed by the host. More...

Typedefs

- typedef struct ARAEditorRendererRefMarkupType ∗ ARAEditorRendererRef

  Reference to the plug-in side representation of a editor renderer (opaque to the host).

Class Documentation

struct ARAEditorRendererInterface

Plug-in interface: editor renderer. The function pointers in this struct must remain valid until the companion API plug-in instance (and accordingly its plug-in extension) is destroyed by the host.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Assigning the playback region(s) for preview while editing

Specify the set of playback region(s) for which the this plug-in instance provides previewing. Hosts can choose to either maintain individual regions, or conveniently maintain regions by specifying region sequences as a shortcut for all regions on those sequences. Behavior is identical in both cases, but hosts should not mix both types of call on a single plug-in instance. Preview should be unambiguous, i.e. there should be only one editor renderer plug-in instance associated with each playback region. During transitions however (such as moving a region from one track to another), this rule may be disregarded temporarily. The host can make these calls while the plug-in is in render-state (”active” in VST3 speak, ”initialized” in Audio Unit speak). Plug-ins must implement a proper bridging to the concurrent render threads. Further, the host may display the UI for the plug-in while making these calls. They can be made both inside or outside any editing or restoration cycles of the associated ARA document controller, but doing them between ARADocumentControllerInterface::beginEditing() ARADocumentControllerInterface::endEditing() when feasible may yield better performance since all updates can be calculated together. The calls shall be made on the same thread that also hosts the document controller. The host is not required to remove any or all playback regions or region sequences before destroying the plug-in. If a plug-in does not implement any audio preview features, it can safely ignore these calls and provide empty implementations.

- void(∗ **addPlaybackRegion**)(ARAEditorRendererRef editorRendererRef, ARAPlaybackRegionRef playbackRegionRef)
- void(∗ **removePlaybackRegion**)(ARAEditorRendererRef editorRendererRef, ARAPlaybackRegionRef playbackRegionRef)
- void(∗ **addRegionSequence**)(ARAEditorRendererRef editorRendererRef, ARARegionSequenceRef regionSequenceRef)
- void(∗ **removeRegionSequence**)(ARAEditorRendererRef editorRendererRef, ARARegionSequenceRef regionSequenceRef)

Editor View Interface (Added In ARA 2.0)

Detailed Description

See kARAEditorViewRole.

Users will often reconfigure the plug-in view through scrolling, zooming, navigating lists of entities, etc. to select the subset of ARA entities that they currently need to view or edit. Those selection features can be implemented in the plug-in (which was the only available solution in ARA 1). However, the host applications already have established user workflows for selecting their representations of the ARA objects. Making those workflows available to the plug-ins is leads to a much more consistent, streamlined user experience. Since the views are implemented through the companion API, there is no matching ARA entity yet. Instead, the companion plug-in instance is used as a controller for its associated view. (Note that while some companion APIs allow for multiple views of a given plug-in used at the same time, this is not recommended when using ARA editor views.) These calls only affect views, not the audio rendering.

They only should be made while the plug-in is showing its UI, or before entering this state (i.e. during GUI setup phase), in order to optimize resource usage. Accordingly, the host should send an update of the selection when (re-)opening an ARA plug-in view. These calls also may be made while changes are being made to the model graph (i.e. inside of pairs of ARADocumentControllerInterface::beginEditing() and ARADocumentControllerInterface::endEditing()).

Plug-in developers using C++ ARA Library can implement the ARA::PlugIn::EditorViewInterface, or extend the already implemented ARA::PlugIn::EditorView class as needed. For host developers this interface is wrapped by the ARA::Host::EditorView.

Classes

- struct ARAViewSelection

  Host generated ARA view selection. More...
- struct ARAEditorViewInterface

  Plug-in interface: view controller. The function pointers in this struct must remain valid until the document controller is destroyed by the host. More...

Typedefs

- typedef struct ARAEditorViewRefMarkupType * ARAEditorViewRef

  Reference to the plug-in side representation of a editor view (opaque to the host).

Class Documentation

struct ARAViewSelection

Host generated ARA view selection.

Public Attributes

- ARASize structSize

  Size-based struct versioning - see Versioned structs.
- ARASize playbackRegionRefsCount

  Length of playbackRegionRefs.
- const ARAPlaybackRegionRef * playbackRegionRefs

  Variable-sized C array listing the explicitly selected playback regions. The list may be empty, in which case count should be 0 and the pointer NULL. If the plug-in requires a playback region selections in its view, it can derive an implicit playback region selection from the region sequence selection and the time range in that case.
- ARASize regionSequenceRefsCount

  Length of regionSequenceRefs.
- const ARARegionSequenceRef * regionSequenceRefs

  Variable-sized C array listing the explicitly selected region sequences. The list may be empty, in which case count should be 0 and the pointer NULL. If the plug-in requires a region sequence selections in its view, it can derive an implicit region sequence selection from the playback region selection in that case.
- const ARAContentTimeRange * timeRange

  Pointer to the explicitly selected time range. This pointer can be NULL to indicate that selection command does not include an explicit time range selection. If the plug-in requires a time range selection in its view, it can derive an implicit time range selection from the playback region selection, or if that isn't provided either it can evaluate the list of all playback regions of the selected region sequences.

struct ARAEditorViewInterface

Plug-in interface: view controller. The function pointers in this struct must remain valid until the document controller is destroyed by the host.

Size-based struct versioning

See Versioned structs.

- ARASize **structSize**

Host UI notifications

- void(∗ notifySelection)(ARAEditorViewRef editorViewRef, const ARAViewSelection ∗selection)

  Apply the given host selection to all associated views. This ARA 2.0 addition allows hosts to translate the effects of their selection implementation to a selection of ARA objects that can be interpreted by the plug-in. The command is not strict a setter for the given selection state, it rather notifies the plug-in about relevant user interaction in the host so that it can adopt in whatever way provides the best user experience within the context of the given plug-in design. A plug-in may e.g. implement scrolling to relevant positions, select the playback regions in its inspector(s), or filter the provided objects further depending on user settings. The plug-in also remains free to modify its internal selection at any time through its build-in UI behavior. For example, a plug-in may design its UI around individual regions being edited, or around entire region sequences. Melodyne features both modes, switchable by the user. When editing an individual region, it will always pick the "most suitable" region from the selection, and ignore the other selected regions. Melodyne also implements various ways to switch the editable region sequence(s) or playback region at any time independently from the host selection, as well as a user option to temporarily ignore host selection changes to "pin" the current selection. The same "loose" coupling applies on the host side: the selection sent by the host is not necessarily equal to its actual selection, but rather the best representation of the users' intent. For example, the user may be able to select entities that are not directly represented in the ARA API, but relate to a set of playback regions in some meaningful way. In that case, those regions may be sent as selection. The selection command includes various optional entities that describe the selection on the host side, such as playback regions or region sequences. Host will only explicitly provide those objects that best describe their current selection, but plug-ins can often derive other selections from that, such as calculating a region selection based on sequence selection and time range. Most hosts offer both an object-based selection which typically centers around selected arrange events (playback regions) and tracks (region sequences) versus a time-range based selection that is independent of the arrange events but typically also includes track selection. Both modes shall be distinguished by providing a time range only in the latter case. Some hosts even allow to select multiple time ranges, this should be expressed by sending their union range across the API. For an object-based selection, the time range remains NULL, and plug-ins can calculate an implicit time range from the selected playback regions if desired. Arrange event selection may linked to track selection in a variety of ways. For example, selecting a track may select its events (often filter by playback cycle range), and selecting an event may select its associated track as well. The ARA selection must be always updated to reflect any such linking accordingly. In each object list, objects are ordered by importance, with the most relevant, "focused" object being first. So if a plug-in supports only a single region selection, it should use the first region. If it can only show a single region sequence at a time, it should use the first selected sequence, or if no sequences are included in the selection fall back to the sequence of the first selected region, etc. Each new selection call describes a full selection, i.e. replaces the previous selection. An empty selection is valid and should be communicated by the host, since some plug-ins may need to evaluate this in special cases. Being selected does not affect the life time of the objects - selected objects may be deleted without updating the selection status first. Note that a pointer to this struct and all pointers contained therein are only valid for the duration of the current call receiving the pointer - the data must be evaluated/copied inside the call, and the pointers must not be stored anywhere.

- void(∗ notifyHideRegionSequences)(ARAEditorViewRef editorViewRef, ARASize regionSequenceRefsCount, const ARARegionSequenceRef regionSequenceRefs[ ])

  Reflect hiding of region sequences in all associated views. Some hosts offer the option to hide arrange tracks from view, so that they are no longer visible, while still being played back regularly. This can be communicated to the plug-in so that it follows suite. Each call implicitly unhides all previously hidden region sequences, so calling this with an empty list makes all sequences visible. The regionSequenceRefs

pointer is only valid for the duration of the call, it must be evaluated inside the call, and the pointer must not be stored anywhere. It should be NULL if regionSequenceRefsCount is 0.

Deprecated: Plug-In Extension Interface.

Detailed Description

This interface was used before ARA 2.0 defined dedicated plug-in roles. It is only to be implemented when ARA 1 backwards compatibility is desired. An ARA 1 call to set/removePlaybackRegion() in this interface is equivalent to calling both set/removePlaybackRegion() in ARAPlaybackRendererInterface and add/removePlaybackRegion() in ARAEditorRendererInterface. To some extend ARA 1 also uses this to for tasks now associated with ARAEditorViewInterface: opening the UI of an ARA 1 plug-in instance is interpreted as selection of the playback region set via this interface.

ARA Audio File Chunks

Detailed Description

To allow for distributing persistent ARA audio source state information together with the underlying audio file in a way that is transparent to the plug-ins and can be supported by all hosts, ARA 2.0 defines a format for embedding such states into standardized audio file chunks. From there, they can be imported into any ARA document using Partial Document Persistency.

The most obvious use case for this is that it enables audio content providers to ship audio files with properly validated, ready-to-use audio source analysis for multiple plug-ins (or incompatible versions of a plug-in if needed). For example, this allows for loading polyphonic audio loops into Melodyne without time-consuming analysis and very quickly adjusting them to follow the song key and chord progression, making the published audio material much more versatile to use in various productions. Other scenarios where such file chunks are used include exporting data from one plug-in to another, or adding ARA objects to a host document via dragging and dropping audio files from a plug-in that either generates these files on the fly (e.g. export of layers in SpectraLayers) or copies them from a built-in sound library.

The ARA chunk should be evaluated by the host both when adding a new audio file to the arrangement and when applying a new/different ARA plug-in for a region/file already used in the arrangement. Note that after loading the data, ARA content readers can be used to extract more information about the audio source - such as tempo map, time and key signatures, etc.

Plug-in vendors shall optimize the encoding of the audio source state information for audio file chunks according to very different criteria compared to encoding the same state for regular ARA song document archives: The audio file states are going to be widely distributed and will be used over a long period of time in very different contexts, whereas song documents are typically only used on a single machine for a rather short time. Audio file archives therefore should emphasize small data size over en-/decoding speed - encoding is only done once, and decoding only happens for a single audio source at a time (compared to hundreds of audio sources in a typical song archive). Even more important, audio file archives are likely going to be used across a wide range of products versions and shall be stable across a long time. The encoding should therefore be as much backwards compatible as possible, potentially even using different encoding based on the current state of the audio source: if e.g. a particular non-backwards compatible feature of the plug-in is not used in the given state, the plug-in can choose an older format to store the data than if that particular feature was utilized. For these reasons, audio file chunks will typically not use the ARAFactory::documentArchiveID but instead one of the IDs listed in ARAFactory::compatibleDocumentArchiveIDs.

Creating audio file chunks may not be meaningful nor supported for any given plug-in. If for example the plug-in does not perform any costly analysis and has no relevant editable audio source state, there is no reason to create audio file chunks for it. Therefore, creating such chunks is currently done only through dedicated authoring tools (such as Melodyne's standalone version) and not directly available in ARA host applications.

Covering both AIFF and WAVE formats, ARA stores its data by extending iXML chunks as specified here: http://www.ixml.info Inside the iXML document, there's a custom tag <ARA> that encloses a dictionary of audio source archives, encoded as array tagged <audioSources>. Each entry in the array is intended for a different plug-in (or incompatible version fo a plug-in) and contains the tag <documentArchiveID> which also functions as the key for the dictionary, and associated data which includes the actual binary archive and meta information, for example:

```
   00
<ARA>
 <audioSources>
 <audioSource>
 <documentArchiveID>com.celemony.ara.audiosourcedescription.13</documentArchiveID>
 <openAutomatically>false</openAutomatically>
 <suggestedPlugIn>
 <plugInName>Melodyne</plugInName>
 <lowestSupportedVersion>5.0.0</lowestSupportedVersion>
 <manufacturerName>Celemony</manufacturerName>
 <informationURL>https://www.celemony.com</informationURL>
 </suggestedPlugIn>
 <persistentID>59D4874F-FA5A-4FE8-BAC6-0E8BC5F6184A</persistentID>
 <archiveData>TW9pbiBEdQ==</archiveData>
 </audioSource>
 <!-- ...  potentially more archives keyed by different documentArchiveIDs here ...  -->
 </audioSources>
</ARA>
```

Variables

- constexpr auto kARAXMLName_ARAVendorKeyword { "ARA" }

  Name of the XML element that contains the vendor-specific iXML sub-tree for ARA.
- constexpr auto kARAXMLName_AudioSources { "audioSources" }

  Name of the XML element that contains the dictionary of audio source archives inside the ARA sub-tree.
- constexpr auto kARAXMLName_AudioSource { "audioSource" }

  Name of each XML element inside the dictionary of audio source archives.
- constexpr auto kARAXMLName_DocumentArchiveID { "documentArchiveID" }

  Name of the XML element inside an audio source archive that acts as unique dictionary key for the list of audio source archives and identifies the opaque archive content. string value, see ARAFactory::documentArchiveID and ARAFactory::compatibleDocumentArchiveIDs.
- constexpr auto kARAXMLName_OpenAutomatically { "openAutomatically" }

  Name of the XML element inside an audio source archive that indicates whether the host should immediately load the archive data into a new audio source object and create an audio modification and playback region for it, or else import the audio file without ARA initially and only load the ARA archive later on demand when the user manually requests it by adding a matching plug-in. boolean value ("true" or "false").
- constexpr auto kARAXMLName_SuggestedPlugIn { "suggestedPlugIn" }

  Name of the XML element inside an audio source archive that provides user-readable information about the plug-in for which the archive was originally created. This can be used for proper error messages, e.g. if openAutomatically is true but no plug-in compatible with the archive's given documentArchiveID is installed.
- constexpr auto kARAXMLName_PersistentID { "persistentID" }

  Name of the XML element inside an audio source archive that encodes the persistent ID that was assigned to the audio source when creating the archive. When loading the archive, the plug-in will use this persistent ID to find the target object to extract the state to. string value, see ARAAudioSourceProperties::persistentID, see ARARestoreObjectsFilter.
- constexpr auto kARAXMLName_ArchiveData { "archiveData" }

  Name of the XML element inside an audio source archive that encodes the actual binary data of the archive in Base64 format, with the possible addition of line feeds as allowed by MIME. Note that it is preferred to encode without line feeds, but decoders must handle both cases. string value, see ARAArchivingControllerInterface, see https://tools.ietf.org/html/rfc4648.
- constexpr auto kARAXMLName_PlugInName { "plugInName" }

Name of the XML element inside a suggested plug-in element that encodes the plug-in name as string.

- constexpr auto kARAXMLName_LowestSupportedVersion { "lowestSupportedVersion" }

  Name of the XML element inside a suggested plug-in element that encodes the minimum version of the plug-in that is compatible with this archive as string.
- constexpr auto kARAXMLName_ManufacturerName { "manufacturerName" }

  Name of the XML element inside a suggested plug-in element that encodes the plug-in manufacturer as string.
- constexpr auto kARAXMLName_InformationURL { "informationURL" }

  Name of the XML element inside a suggested plug-in element that encodes the plug-in information URL as string.

Companion APIs

Detailed Description

Audio Unit

Detailed Description

ARA support for Audio Units is mostly based on private Audio Unit properties. The input/output pattern for the associated property data structs is similar to struct AudioUnitParameterStringFromValue in the Audio Unit API. To be able to detect potential collisions if some host or Audio Unit uses the same private property for non-ARA communication, the actual ARA data is embedded into a property data struct that also contains an ARA magic number. For this magic number to work properly for all host/plug-in combinations, it's crucial that hosts set it on input and check it after output, and that plug-ins check it before writing to the property struct.

Classes

- struct ARAAudioUnitFactory

  Property data for kAudioUnitProperty_ARAFactory. *More...*
- struct ARAAudioUnitPlugInExtensionBinding

  Property data for kAudioUnitProperty_ARAPlugInExtensionBindingWithRoles. *More...*

Macros

- #define kARAAudioComponentTag "ARA"

  Integration with the system-wide Audio Unit cache. Since the introduction of the Audio Component APIs in macOS 10.10 Yosemite, a system-wide cache of all installed Audio Units allows hosts to defer opening any Audio Unit until the user explicitly requests this. To support this optimization also when ARA is used, plug-ins must be marked as ARA-compatible at the Audio Component level by including the tag "ARA" in their Info.plist file, like in this example:

Class Documentation

struct ARAAudioUnitFactory

Property data for kAudioUnitProperty_ARAFactory.

Public Attributes

- OSType inOutMagicNumber

  Input/output: token to identify the property is actually used for ARA.
- const ARAFactory * outFactory

  Output: pointer to the factory associated with the Audio Unit.

struct ARAAudioUnitPlugInExtensionBinding

Property data for kAudioUnitProperty_ARAPlugInExtensionBindingWithRoles.

Public Attributes

- OSType inOutMagicNumber

  Input/output: token to identify the property is actually used for ARA.
- ARADocumentControllerRef inDocumentControllerRef

  Input: document controller of the model graph that the Audio Unit shall be bound to.
- const ARAPlugInExtensionInstance * outPlugInExtension

  Output: the entry point for the plug-in.
- ARAPlugInInstanceRoleFlags knownRoles

  Input: all roles that the host considered in its implementation and will explicitly assign to some plug-in instance(s). Being an ARA 2.0 addition, this field is only available if kAudioUnitProperty_ARAPlugInExtensionBindingWithRoles is used.
- ARAPlugInInstanceRoleFlags assignedRoles

  Input: roles assigned to this specific plug-in instance. Being an ARA 2.0 addition, this field is only available if kAudioUnitProperty_ARAPlugInExtensionBindingWithRoles is used.

Macro Definition Documentation

kARAAudioComponentTag

```
#define kARAAudioComponentTag "ARA"
```

Integration with the system-wide Audio Unit cache. Since the introduction of the Audio Component APIs in macOS 10.10 Yosemite, a system-wide cache of all installed Audio Units allows hosts to defer opening any Audio Unit until the user explicitly requests this. To support this optimization also when ARA is used, plug-ins must be marked as ARA-compatible at the Audio Component level by including the tag ”ARA” in their Info.plist file, like in this example:

```
   00
<key>AudioComponents</key>
<array>
 <dict>
 ...
 <key>manufacturer</key>
 <string>Demo</string>
 <key>subtype</key>
 <string>Test</string>
 <key>type</key>
 <string>aufx</string>
 ...
 <key>tags</key>
 <array>
 <string>Effect</string>
 ...
 <string>ARA</string>
 </array>
 </dict>
</array>
```

Enumeration Type Documentation

anonymous enum

```
anonymous enum
```

Enumerator

kARAAudioUnitMagic	This value must be placed in the magicNumber fields of the ARA properties.

anonymous enum

```
anonymous enum
```

Enumerator

kAudioUnitProperty_ARAFactory	kAudioUnitScope_Global, read-only, property data is ARAAudioUnitFactory: query the ARA factory associated with the given plug-in.

anonymous enum

```
anonymous enum
```

Enumerator

kAudioUnitProperty_ARAPlugInExtensionBinding	kAudioUnitScope_Global, read-only, property data is ARAAudioUnitPlugInExtensionBinding: bind the Audio Unit instance to an ARA document controller, switching it from ”normal” operation to ARA mode, and exposing the ARA plug-in extension. Note that since ARA 2.0, this property has been deprecated and replaced with kAudioUnitProperty_ARAPlugInExtensionBindingWithRoles. This deprecated call is equivalent to the new call with no known roles set, however all ARA 1.x hosts are in fact using all instances with playback renderer, edit renderer and editor view role enabled, so plug-ins implementing ARA 1 backwards compatibility can safely assume those three roles to be enabled if this call was made. Same call order rules as kAudioUnitProperty_ARAPlugInExtensionBindingWithRoles apply.

anonymous enum

```
anonymous enum
```

Enumerator

kAudioUnitProperty_ARAPlugInExtensionBindingWithRoles	ARA 2 extended version of kAudioUnitProperty_ARAPlugInExtensionBinding: kAudioUnitScope_Global, read-only, property data is ARAAudioUnitPlugInExtensionBinding: bind the Audio Unit instance to an ARA document controller, switching it from "normal" operation to ARA mode with the assigned roles, and exposing the ARA plug-in extension. This may be done only once during the lifetime of the Audio Unit, before initializing it via kAudioUnitInitializeSelect or setting its state via the properties ClassInfo, PresentPreset or CurrentPreset or before creating any of the custom views for Audio Unit. The ARA document controller must remain valid as long as the plug-in is in use - rendering, showing its UI, etc. However, when tearing down the plug-in, the actual order for deleting the Audio Unit and for deleting ARA document controller is undefined. Plug-ins must handle both potential destruction orders to allow for a simpler reference counting implementation on the host side.

Audio Unit V3

Detailed Description

Experimental support for Audio Unit v3 (App Extension)

Classes

- protocol <ARAAudioUnit>

  Protocol to be implemented by ARA-compatible subclasses AUAudioUnit. This protocol may or may not be replaced with the AUMessageChannel based communication defined below. More...

Macros

- #define kARAAudioComponentTag "ARA"

  Integration with the system-wide Audio Unit cache. To allow hosts to defer opening any Audio Unit v3 App Extensions until the user explicitly requests it, the system caches various information about the Audio Unit. To support this optimization also when ARA is used, plug-ins must be marked as ARA-compatible at the Audio Component level inside its NSExtensionAttributes by including the tag "ARA" in its Info.plist file, like in this example:
- #define ARA_AUDIOUNIT_FACTORY_CUSTOM_MESSAGES_UTI /*ARA_DRAFT*/ @"org.ara-audio.audiounitmessages.factory"

UTIs for ARA message protocols used for [AUAudioUnit messageChannelFor:] (added in macOS 13) The message channel for ARA_AUDIOUNIT_DOCUMENTCONTROLLER_CUSTOM_MESSAGES_UTI should only be obtained and configured once, it will also be used for all document controller communications based on the factories. The message channel for ARA_AUDIOUNIT_PLUGINEXTENSION_CUSTOM_MESSAGES_UTI on the other hand should be managed per ARA-enabled Audio Unit plug-in instance. This API may eventually replace the above ARAAudioUnit <NSObject> draft.

Class Documentation

protocol ARAAudioUnit-p

Protocol to be implemented by ARA-compatible subclasses AUAudioUnit. This protocol may or may not be replaced with the AUMessageChannel based communication defined below.

Inherits <NSObject>.

Macro Definition Documentation

kARAAudioComponentTag

```
#define kARAAudioComponentTag "ARA"
```

Integration with the system-wide Audio Unit cache. To allow hosts to defer opening any Audio Unit v3 App Extensions until the user explicitly requests it, the system caches various information about the Audio Unit. To support this optimization also when ARA is used, plug-ins must be marked as ARA-compatible at the Audio Component level inside its NSExtensionAttributes by including the tag ”ARA” in its Info.plist file, like in this example:

```
    00
<key>NSExtension</key>
<dict>
<key>NSExtensionAttributes</key>
 <dict>
 <key>AudioComponentBundle</key>
 <string>your.identifier.goes.here</string>
 <key>AudioComponents</key>
 <array>
 <dict>
 ...
 <key>manufacturer</key>
 <string>Demo</string>
 <key>subtype</key>
 <string>Test</string>
 <key>type</key>
 <string>aufx</string>
 ...
 <key>tags</key>
 <array>
 <string>Effects</string>
 ...
 <string>ARA</string>
 </array>
 </dict>
 ...
```

VST3

Detailed Description

Classes

- class ARA::IMainFactory

Interface class to be implemented by an object provided by the VST3 factory. The host can use the VST3 factory to directly obtain the ARA factory, which allows for creating and maintaining the model independently of any IAudioProcessor instances, enabling tasks such as automatic tempo detection or audio-to-MIDI conversion. For rendering and editing the model however, there must be an associated IAudioProcessor class provided in the same binary. This match is usually trivial because there typically is only one such class in the binary, but there are cases such as WaveShell where multiple plug-ins live in the same binary, and only a subset of those plug-ins support ARA. In this scenario, the plug-in must use the same class name for the matching pair of ARA::IMainFactory and IAudioProcessor classes - this enables the host to quickly identify the matching pairs without having to create instances of all the IAudioProcessor classes to query their IPlugInEntryPoint::getFactory ()->factoryID to perform the matching. More...

- class ARA::IPlugInEntryPoint

  Interface class to be implemented by the VST3 IAudioProcessor component (kVstAudioEffectClass). More...

- class ARA::IPlugInEntryPoint2

  ARA 2 extension of IPlugInEntryPoint. More...

Macros

- #define kARAMainFactoryClass "ARA Main Factory Class"

  Class category name for the ARA::IMainFactory.

Class Documentation

class ARA::IMainFactory

Interface class to be implemented by an object provided by the VST3 factory. The host can use the VST3 factory to directly obtain the ARA factory, which allows for creating and maintaining the model independently of any IAudioProcessor instances, enabling tasks such as automatic tempo detection or audio-to-MIDI conversion. For rendering and editing the model however, there must be an associated IAudioProcessor class provided in the same binary. This match is usually trivial because there typically is only one such class in the binary, but there are cases such as WaveShell where multiple plug-ins live in the same binary, and only a subset of those plug-ins support ARA. In this scenario, the plug-in must use the same class name for the matching pair of ARA::IMainFactory and IAudioProcessor classes - this enables the host to quickly identify the matching pairs without having to create instances of all the IAudioProcessor classes to query their IPlugInEntryPoint::getFactory ()->factoryID to perform the matching.

Inherits Steinberg::FUnknown.

Public Member Functions

- virtual const ARAFactory *PLUGIN_API getFactory ()=0

  Get the ARA factory. The returned pointer must remain valid throughout the lifetime of the object that provided it. The returned ARAFactory must be equal to the ARAFactory provided by the associated IAudioProcessor class through its IPlugInEntryPoint.

class ARA::IPlugInEntryPoint

Interface class to be implemented by the VST3 IAudioProcessor component (kVstAudioEffectClass).

Inherits Steinberg::FUnknown.

Public Member Functions

- virtual const ARAFactory *PLUGIN_API getFactory ()=0

  Get the ARA factory. The returned pointer must remain valid throughout the lifetime of the object that provided it. The returned ARAFactory must be equal to the ARAFactory provided by the associated IMainFactory. To prevent ambiguities, the name of the plug-in as stored in the PClassInfo.name of this class must match the ARAFactory.plugInName returned here.

- virtual const ARAPlugInExtensionInstance *PLUGIN_API bindToDocumentController (ARADocumentControllerRef documentControllerRef)=0

  Bind the VST3 instance to an ARA document controller, switching it from "normal" operation to ARA mode, and exposing the ARA plug-in extension. Note that since ARA 2.0, this call has been deprecated and replaced with bindToDocumentControllerWithRoles (). This deprecated call is equivalent to the new call with no known roles set, however all ARA 1.x hosts are in fact using all instances with playback renderer, edit renderer and editor view role enabled, so plug-ins implementing ARA 1 backwards compatibility can safely assume those three roles to be enabled if this call was made. Same call order rules as bindToDocumentControllerWithRoles () apply.

class ARA::IPlugInEntryPoint2

ARA 2 extension of IPlugInEntryPoint.
Inherits Steinberg::FUnknown.

Public Member Functions

- virtual const ARAPlugInExtensionInstance *PLUGIN_API bindToDocumentControllerWithRoles (ARADocumentControllerRef documentControllerRef, ARAPlugInInstanceRoleFlags knownRoles, ARAPlugInInstanceRoleFlags assignedRoles)=0

  Extended version of bindToDocumentController (): bind the VST3 instance to an ARA document controller, switching it from "normal" operation to ARA mode with the assigned roles, and exposing the ARA plug-in extension. `knownRoles` *encodes all roles that the host considered in its implementation and will explicitly assign to some plug-in instance(s), while* `assignedRoles` *describes the roles that this specific instance will fulfill. This may be called only once during the lifetime of the IAudioProcessor component, before the first call to setActive () or setState () or getProcessContextRequirements () or the creation of the GUI (see IPlugView). The ARA document controller must remain valid as long as the plug-in is in use - rendering, showing its UI, etc. However, when tearing down the plug-in, the actual order for deleting the IAudioProcessor instance and for deleting ARA document controller is undefined. Plug-ins must handle both potential destruction orders to allow for a simpler reference counting implementation on the host side.*

Index