Episode 3: A new API

I wish this was Episode IV...

Introduction

In the RISC OS Open forums, the discussion went on a different path to where I had envisaged it – a new API.

Adding functionality

If an API has some missing functionality, there's generally two options:

1. The API is extended to add the new functionality
2. You create a new API with the additional functionality

Extending an API

Previously, all of the discussions were on extending the existing SoundDMA (and SharedSound) APIs so that they were able to cope with the new functionality (namely sound input, 24-bit and 32-bit formats, multi-channel devices [i.e. greater than 2], and support for multiple and different hardware output mechanisms).

Extending an API is generally preferred, because it means that any existing hardware that is using the API *should* still work (note that the "should" is caveated).

Creating a new API

A new API isn't something that you'd want to do – there are times when it's necessary, and times when you don't want to.

Reasons for a new API

Some reasons for creating a new API (this list is not exhaustive):

- The underlying implementation of the old API is inherently flawed
- The old API is copyrighted, and the owner does not want to make it available
- A new API would make it easier for adopters to use
- The old API is overtly complex, and extending it would be difficult
- The new features would be difficult to implement in the old API
- The old API has features that ought to have been removed a while back

Reasons for extending an API

Some reasons why you'd want to extend an API instead of creating a new one (again, this list is not exhaustive):

- As mentioned above, existing applications should still work; new applications simply use the extensions

- Applications do not need to support multiple APIs if they want to remain backwardly compatible
- The work in creating the new API is significantly more than extending the API

Co-existing APIs

It is entirely possible for a new API to co-exist with an old API – three main approaches can be taken (note that the descriptions of these start off as a generic approach, not specific to RISC OS Sound).

Side-by-side

If the APIs do not “tread on each other’s feet”, then they can simply work at the same time. The application will simply decide which API to use (either because it’s only been written to use one API, a compile-time switch allows the API to be changed, or the user can select the API).

This, unfortunately, is not possible, because the APIs will end up having to use the same hardware.

One-at-a-time

If the new API will prevent the operation of the old API (or vice-versa), then you can make one of them mutually exclusive of the other. The rules for which API has priority could vary, but an option of the following would be most likely:

- Operating system configuration
- First-come-first-served (first API to be used is the one that stays being used)
- Most recent first (last application to start up select API)
- New API preferred (if all applications want to use the old API, then that’s what will be used, but as soon as one wants to use the new API, then that is used instead)

This is probably not desired because it may leave the user confused as to why the applications can’t play nicely with each other, or wonder why things aren’t working.

Shims

Another option if the APIs “collide” is to use shims. This is basically adding in support for the old API in the new API – i.e. the old API still works; it just becomes translated into the new API in a manner which is hidden to the application. Any applications using the old API thinks it’s still using the old API, but it’s actually using the new API.

This means that old applications do not need to be rewritten in order to use the new API. They’ll automagically work.

This is a bit more work, but ultimately allows both new and old applications to work with the hardware.

Do we need a new API for RISC OS Sound?

Before we can answer this, we need to look at what is being proposed for new features in the RISC OS Sound system, and whether an extended API or a new API is applicable.

Sound input capability

RISC OS does not currently have a unified sound input API. This means that it is entirely possible to invent a completely new API, and not worry about having to support existing applications – there aren't any!

Conclusion:

New API is easy

Additional sample formats

The current API only supports 16-bit LPCM sound samples. This was okay when the original API was developed, but the music industry has pushed the higher bit-depth sample formats into the mainstream, and now many low-cost hardware interfaces support higher bit-depths.

In addition, audio files tend to be compressed in order to make them more transportable – this means that LPCM is not the default for the files located on your hard disk. Non-LPCM transfer mechanisms are currently not supported, so a new API would be fine.

Conclusion:

Adding support for higher bit-depth can be done with extending the API, but complications occur when two different applications want to use different bit-depths (as discussed in my earlier document). Since applications exist that are using the older APIs (both of them), then either extending the API, or new API+shim would be the best approach.

Adding support for non-LPCM would be best done with a new API.

Multi-channel devices

Since the current API only supports 2-channel devices, there is no backward compatibility issues for hardware and applications that support more than two channels

Conclusion:

A new API would be fine for more than 2 channels, but to allow old applications to use multi-channel hardware devices (in their left and right channels), a shim can be used.

Multiple, and different hardware interfaces

The current sound system only supports 2-channel 16-bit LPCM interfaces, and there is no API for the sound system to 'talk' to the hardware interfaces as the hardware interface code is closely coupled to the sound system.

Conclusion:

Seeing as there is no API for hardware to interface to the sound system, a new API would be needed anyway.

Adding support for more than 2 channels

Overall conclusion

With the exception of existing applications playing 16-bit LPCM samples, everything could work with a new API, as it's all new functionality. These applications can be catered for with a shim from the old API to the new.

So at the moment, a new API with shims looks good.

Is there anything else that suggests a new API is needed?

Let's look at the reasons behind why you'd want a new API:

The underlying implementation of the old API is inherently flawed

The main failing in the old API is the lack of independent hardware support. Is that a major flaw? To some extent, yes – when you start needing to support hardware that does not support LPCM samples (such as Bluetooth).

Having applications able to support native non-LPCM formats without having to convert them to LPCM would be a benefit – but only if the non-LPCM format happened to match what the hardware was expecting.

Conclusion:

This is not a major stumbling block, so you could extend the old or create a new API.

The old API is copyrighted, and the owner does not want to make it available

This is not the case – the source code is available.

Conclusion:

There is nothing preventing the old API being extended.

A new API would make it easier for adopters to use

The method of a sound program getting data to and from the hardware would not differ much; there may be restrictions on the old, and new restrictions on the new – but fundamentally, it's the same process ("Here's some data", or "Thanks for the data").

Conclusion:

There would be very little difference between extending the old API and creating a new.

The old API is overtly complex, and extending it would be difficult

The old API is actually quite simple. Sort of: there are two APIs, SoundDMA and SharedSound. These are in a "one-at-a-time" relationship. Applications would need to be written to support at least one of these APIs.

Later APIs, such as USB, have been developed, but are implemented as a "side-by-side" approach with the existing sound APIs. This means applications would need to implement these as an additional case.

Conclusion:

A new API + shims would permit all three interfaces to be implemented by the new API.

The new features would be difficult to implement in the old API

Adding support for higher bit-depth samples would be fairly easy to add into the old API, but support for non-LPCM samples and multi-channel interfaces would be quite tricky, and any implementation would effectively be a new API anyway

Conclusion:

A new API is the best way to add in the new features.

The old API has features that ought to have been removed a while back

The 8-bit µ-Law audio sound system is one of those areas that is questionable as to whether it is needed or not. Probably the only application that will stop working is *!Maestro* – is that a big problem?

Having said that, the way it is implemented means it probably has very limited impact in generating a new API.

Conclusion:

Apart from the µ-Law audio, there isn't that much that needs to be removed from the current API, so it does not matter if the API is extended or a new one is created.

Is there anything that suggests extending the API is preferable?

Again, let's look at the reasons for extending the API rather than creating a new one.

...existing applications should still work

By adding shims to a new API, existing applications should work as before with the old API. Shims would need to be created for each API.

Conclusion:

A new API would be okay.

Applications do not need to support multiple APIs if they want to remain backwardly compatible

If a developer wants their application to work on a wider range of computers, then they would need to support both the new API *and* the old API. This is already the case with SoundDMA and SharedSound; SharedSound was a later addition to RISC OS, and not all computers will have it.

However, in order to use the new features being proposed here, a new implementation would need to be done regardless of whether it's encapsulated as a new API, or as extensions. There might be a replication of work (for example, to support 2-channel 16-bit LPCM samples if a developer wanted to support the new API natively as well as the old).

Developers could decide not to implement the old API at all – although that would limit distribution.

Conclusion:

While it is a bit more work on the developer, it is not much compared to the rest of the development – and some of it would be needed in any case.

Therefore a new API would be probably be fine.

The work in creating the new API is significantly more than extending the API

If a new API + shims is the option selected, then this would be more potentially work than simply extending the existing API, since the existing API would not need the shims to be created.

Conversely, it is entirely possible that the new implementation may actually be more efficient with shims (both in code performance, and the source code itself).

Conclusion:

It would be difficult to weigh both approaches without actually trying it. However, I believe that trying to weave the non-LPCM implementation into the LPCM code would be quite complicated, so the argument above regarding ease of feature implementation would trump any argument about the work involved.

If the new work was to *only* support higher bit-depth sample formats and additional LPCM-only hardware, then it could be argued the other way.

What the new API would look like

There are four control areas within the new API – one is completely new, two already exist with limited functionality, and the fourth is embryonic compared to what is envisaged.

Buffer manager

This part of the API is responsible for the main interface with the hardware abstractions, and handles passing audio streams between applications and the HALs.

Control interface

This is responsible for providing controls over the hardware components – such as volume levels, filtering, oversampling and so forth.

Discovery interfaces

This is responsible for keeping track of the HALs that register with it, and allowing applications to find out about the audio capabilities of the system

Codec interfaces

This looks after the codecs that are available on the system. Modules would register support for format translators, and applications and HALs can use them in order to get the data into the format that they desire.

How this looks

An idea of this is shown in figure 1:

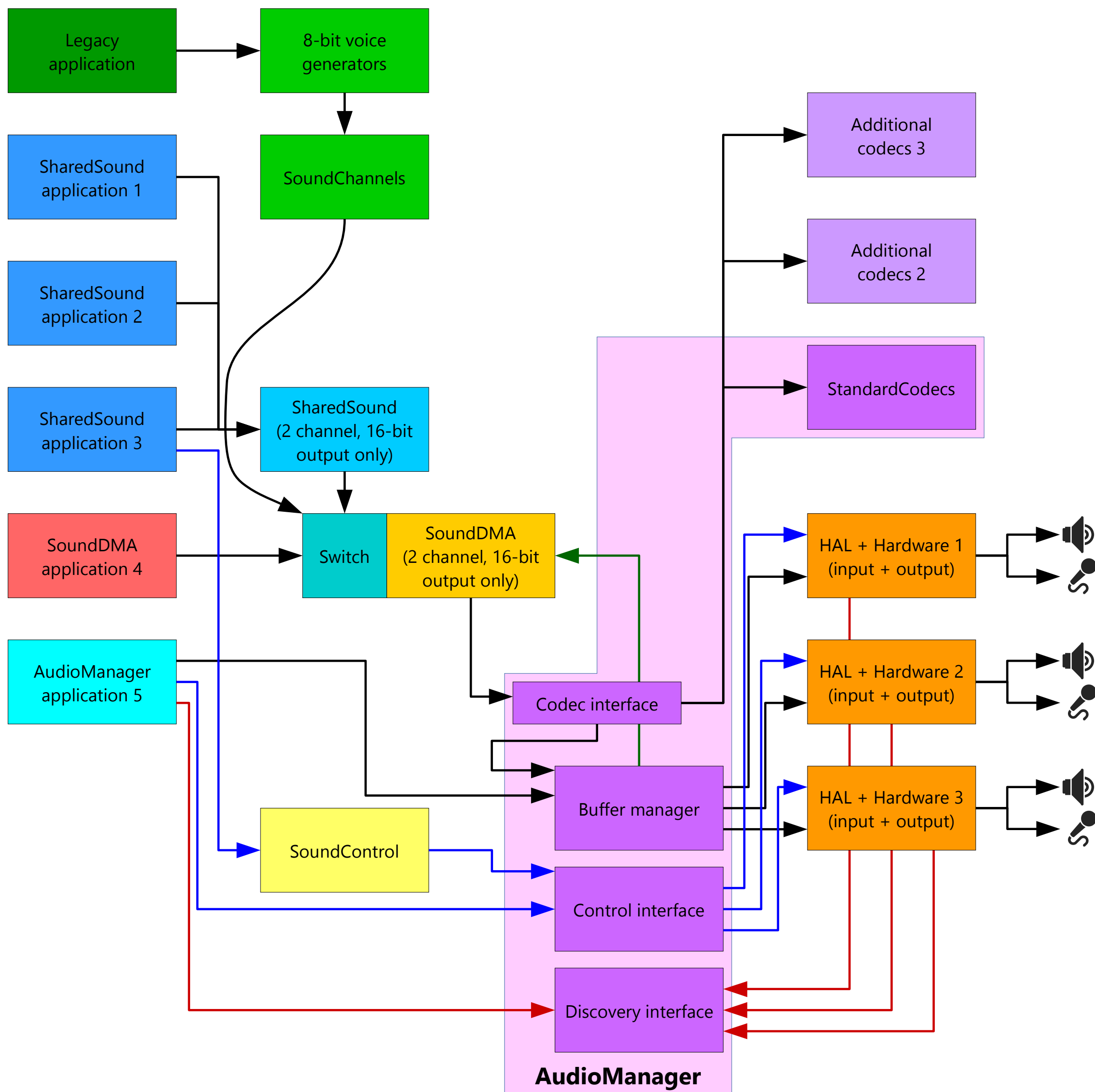

Figure 1: AudioManager block diagram

Note that I have decided to use the “AudioManager” name that (sort-of) cropped up in the RISC OS Open forum discussion. The name in the discussion was “AudioMan”, but I’d prefer the longer name.

API proposals

I would propose the following change in the way RISC OS Sound works as follows:

Buffer sizes

The ability for an application to change the buffer size is removed – this should be a configuration setting.

Although I would note that music creation software would generally want to have the buffer size to be as small as possible in order to reduce latency. This will need to be considered.

Justification:

- Having multiple applications wanting to change the buffer size is a nightmare in sharing.
- Some hardware may function more efficiently with specific buffer sizes, and it would be best to align with the hardware
- Buffer sizes can somewhat nebulous when it comes to non-LPCM data

Application implications:

- All applications will need to fill the sound buffer with data according to the available space – they cannot make assumptions that “the buffer has 256 samples available”.
- The only applications which may require a certain buffer sizes are going to be SoundDMA/ LinearHandler applications – hopefully they are not used that often!

HAL implications:

- HALs will need to identify the ideal buffer size(s) to the discovery interface.
- If a change in buffer size is possible and requested, then the HAL will identify if the new buffer size is suitable.

LPCM format

Having to support 16-bit, 24-bit and 32-bit LPCM data in an application, buffer manager and HAL requires the near replication of code to fill buffers. Therefore I am proposing that the 16-bit and 24-bit LPCM formats are not directly supported in the new API. LPCM buffers are 32-bit only.

Justification:

- Applications need to fill buffers, and merge with previous data. Different code is needed for each format – although the code would be fairly similar in nature. Having only a single buffer fill code would simplify development.

Application implications:

- Applications only require a single piece of code to be written to support the new API, regardless of the encoding of the data, resulting in a simplification of development.
- A developer does not need hardware that supports all of the formats in order to ensure their application runs correctly.
- SoundDMA would ultimately need to create the data in 32-bit format. This can be done via the codec interface (discussed later).

HAL implications:

- HALs would need to accept 32-bit data; if data in another format is required, then they would either need to re-code the data themselves, or use the codec interface to do the work for them (although converting from 32-bit to 24-bit or 16-bit is trivial enough for the codec to do it).
- DMA can only be used on the filled buffer if the output format is 32-bit, which means it is likely to be less efficient. However, the savings on the buffer filling and merging would almost certainly outweigh this.

Shared sound-like implementation

The ability for the sound system to be shared by multiple applications is something that a lot of people feel strongly about. Therefore, I propose that this new API is inherently shared.

Justification:

- Without a shared interface, the only method that would be available for applications to share the sound would be via the (old) SharedSound interface – which may one day end up no longer supported.
- Dedicated applications can still be developed – although this ought to be discouraged.
- Parallel processing could be easily taken advantage of – for example, one CPU could be preparing the SoundDMA 16-bit buffer, whilst another is calling the AudioManager applications to fill in their data. When complete, the two data streams are merged. It would also be possible for the AudioManager applications to fill their data on individual CPUs, and merged later on.

Application implications:

- Applications must perform merging of data from the previous application-level
- SoundDMA can be implemented as a pseudo-application, providing its data into the next application into the chain
- Non-LPCM streams from applications will need to be converted to LPCM before merging. Applications should use the codec interface to do this.

HAL implications:

- None

Sample format identification

For ease of implementation, it is highly advantageous to make the sample format fit into a single word – however, it should not be a restrictive format.

USB sample format descriptors

The USB audio data format specification [USB001] defines several different types of format type descriptors.

Type I formats allow one of five different audio formats – all of them PCM in some way. They are:

- PCM
- PCM8 (8-bit wave format)
- IEEE_FLOAT (single precision)
- ALaw
- µLaw (but note this is not RISC OS µLaw)

All of these have an 8-bit number of channels, 8-bit subframe size, and an 8-bit resolution bitfields. Note that an 8-bit sample frequency type also exists – but this would impact the sample rate, rather than the format of the data itself.

Type II formats are used for non-PCM encoded audio data, with data that is normally framed.

The formats available are:

- MPEG (with a 16-bit capabilities and an 8-bit features bitfields)
- AC-3 (with a 32-bit BSID [BitStream IDentification] mode)

The Type III format is the same as Type I, except it is set up for two-channel 16-bit PCM data, and the data is framed.

Appendix A of [USB001] gives a "format tag" used to identify the formats – this is a 16-bit value, and can be used for the top-16 bits of the format identifier.

For example, a PCM stream is given the format tag of &0001. MPEG as 0x1001.

Note that the appendix seems to indicate that Type III can be used for other purposes – for example, MPEG-1 Layer 2/3 ("MP3") is given the value &2002 (although this would be framed as IEC1937).

Encoding of formats

PCM needs to have additional parameters – namely the number of channels, and the bits per sample.

In addition, knowing where each of the channels are located would be useful when converting from a 5-channel stream to a 7-channel stream – USB does have the concept of a "channel cluster" [USB002], where each channel is at a specific location.

This means that there would be too many bits to encapsulate all of this within a 32-bit word. Therefore a shorter approach is needed.

Shorter format encoding

In order to allow a quick check of standard formats, and extensibility, then one method is to allow both pointers to extended values, and discrete values.

For example, if the sample format is aligned to a word boundary, then the format is an extended format descriptor that the value points to in memory. If it is not word aligned, then it is a discrete value.

The bottom 8-bits are used as the basic format identifier (when they are not a multiple of 4).

LPCM data (format &01)

This is encoded as &ccccbn01, where n is the number of channels (1-15, 0 reserved), b is the number of *bytes* per sample, and cccc is the channel configuration, as per page 34 of [USB001]:

- Bit 0 Left front
- Bit 1 Right front
- Bit 2 Centre front
- Bit 3 Low frequency enhancement
- Bit 4 Left surround
- Bit 5 Right surround

And so forth. Note that bits 12-15 are reserved.

Any value that does not make sense is reserved – but note that the number of channel cluster bits that are set does not need to equal the number of channels (see page 35 of [USB001] for details on how this works for USB in Example 2).

For example, a 2-channel 16-bit LPCM sample on left and right would have the encoding &00032201.

A 5-channel 32-bit LPCM sample on left, right, centre, left surround and right surround would be &00374501.

Note that the format for the internal buffers will always be 32-bit – although the number of channels can vary.

MPEG data (format &02)

This is encoded as &dddddd01 – the values for dddddd will need to be defined at a later date.

It is possible that it starts with the number of channels in the data (which will be 2-channels for MPEG-1, and 5.1 channels for MPEG-2), with the rest denoting more information regarding the MPEG format (such as MPEG-1, MPEG-2, …), and the profile being used (MPEG-2 Layer 7 / MPEG-4).

AAC data (format &03)

This is encoded as &dddddd03 – the values for dddddd will need to be defined at a later date.

It is possible that it starts with the number of channels in the data (up to 48 channels are supported in AAC), with the rest denoting more information regarding the encoding of the data (such as the profile being used).

Bluetooth data (format &05)

This is encoded as &dddddd05 – the values for dddddd will need to be defined at a later date.

The A2DP codec will consume 8 bits (at least 14 are defined [VAL001]; there ought to be more for expansion). The remainder may be used depending on the codec being used, such as the bitpool size for SBC.

Extended formats (word-aligned values)

This will be determined at a later date. Code that is not expecting to deal with any extended format can simply ignore these values., as they will not be supported by the code.

Note that no extended format value will be used to represent one that can be represented as a discrete value (for example, a 16-bit LPCM stereo sample will **always** be represented as &xxxx2201).

Changes to existing modules

Some modules will need changing to integrate with the new API.

SoundDMA

SoundDMA is now a partial shim – it still needs to have some functionality within it, but no longer interacts with hardware – or even HALs.

The buffer manager will signal to SoundDMA that it's ready for some more data (with a 'standard' buffer size, this would be based on the HAL providing the 'fill me' signal). At this point, SoundDMA will present applications with a 2-channel 16-bit buffer that they will fill as before (either directly as a LinearHandler, or indirectly through SharedSound).

Once that process has been completed for a buffer, then the data goes through the codec interface to convert it to 32-bit LPCM data (in the more usual audio format).

AudioManager applications can then merge their data with this data, which is then passed to the HAL in order to populate its data.

Applications using SoundDMA directly should require no changes.

SharedSound

This module should not need changing much – if at all – as it's already implemented as a LinearHandler in SoundDMA.

Applications registering with SharedSound should also need no modifications.

SoundControl

This becomes a thin shim to AudioManager to pass controls through to the HAL via the module.

Applications using SoundControl should need no changes.

Codec interface: translating formats

Note that codec is normally short for "Compression / Decompression" – I prefer to use "codec" rather than "translation", or "encodec" ("Encoder / Decoder"). If someone has a better idea, then let me know!

The codec interface is one of the new aspects of this work – it allows the translation of data from one format to another. Each format is given a unique identifier, and modules register with AudioManager with a pair of identifiers which essentially state "I have code that can translate from format A to format B".

AudioManager keeps track of this list, and if an application needs to translate from A to B, then it makes a call to AudioManager to request the codec's interface in order to perform the translation.

Converting data

In order to convert data, eight basic pieces of information are required:

- Input sample format
- Input sample rate
- Input data
- Input data length
- Output sample format
- Output sample rate
- Output data
- Output data length

A ninth piece of data may also be required – which is any context that is needed to continue the conversion (for example, information from an earlier conversion). This would not be needed for some of the simpler formats.

And if the volume level needs to be changed, or mono/stereo mix, then the mixer settings would need to be provided.

When converted, the amount of input data used, and output data filled would be returned back to the application. A flag would also be used to indicate that the output data is full (this can be inferred from sample formats where the output buffer is a known size, but for other formats, the amount of data in the output buffer may be variable).

Applications must fill the buffer until the 'fill' flag is returned.

Audio capture would be done in a similar fashion (except there will be an 'empty' flag to denote there is no more data that can be read).

Sample rate conversions

In addition to what formats the codecs support, they will need to provide an indicator as to whether they can perform sample rate conversion at the same time, or if they cannot perform this feature at the same time.

A codec may also want to offer a conversion only if the sample rates are different – for example, converting 2-channel LPCM 44.1kHz formatted data into a 2-channel LPCM 44.1kHz format is not necessary, but converting to a 2-channel LPCM 48kHz format is necessary.

Mixer controls

In some cases, having a codec support merging of data with existing data is either not feasible, or not desirable.

For example the final stage for converting data to a Bluetooth stream does not require the sound be merged with anything else, so this would not require any mixer controls.

In addition, audio input codecs do not need any software mixer support, so these can indicate no support for mixer.

Therefore codecs need to indicate that they support software mixer controls or not.

Out-of-boundary buffered data

In some cases, the number of samples contained within an input buffer and an output buffer may be different lengths – in which case some contextual information may be needed to store the discrepancy.

Codec chains

A codec may not exist to translate from format A to format B – however, codecs may exist to translate from format A to format C, and from format C to format B. This would not be as optimal as going directly from A to B, but would at least work. Later on, a codec that translates directly from A to B could be written.

An example would be a basic MP3 decoder. The MP3 decoder can only translate a 44.1kHz input stream to a 44.1kHz LPCM stream. However, the audio hardware is running at 96kHz. This means that a second translator would be needed to convert the 44.1Khz sample rate to 96kHz.

These codec chains could be either automatically generated by AudioManager (the application simply asks for the codec's entry points, and these are constructed at run-time), or the application could be given the codec chain that it would need to negotiate with.

Note that any codec in the chain may require its own contextual data, so the initialisation of this chain would need to allocate memory to all of them.

Buffer usage in the chain

In a non-chained codec, the input data is merged directly into the output data. In a chain, the intermediary steps do not need any form of merging – it is only the final stage where merging may be required (if the format permits it).

Note that if this is the first of the audio playback routines, then it would need to copy the data, rather than merging it.

The memory used for the intermediary stages can be used by subsequent stages – if there is anything that is needed later on, then it should be stored in the contextual data memory block.

Implicit codecs

In some cases, it might be desireable to have some implicit codecs in the chain.

For example, a 5-channel LPCM audio hardware output would want to have its data in a 5-channel format – however, if 3 applications are providing 2-channel output, and a fourth is generating 5-channel output, then it is more optimal to merge the 2-channel outputs together, and then do a conversion to 5-channel that gets merged with the fourth application's output, rather than have each of the three applications' data converted to 5-channel which are then merged together.

In figure 2, the green codec is the implicit one (note that this is a simplified diagram):

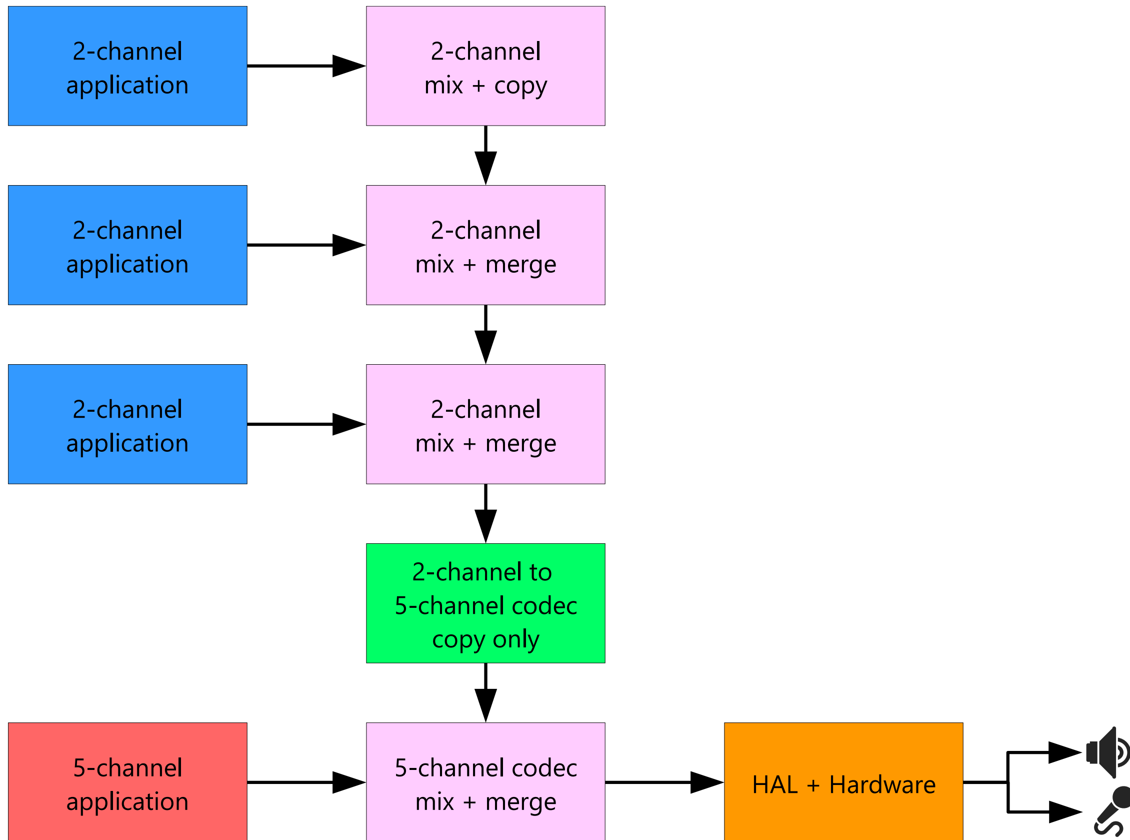

Figure 2: Implicit codec

The control mechanism for this would be quite complex (especially if multiple conversions are required), so the initial implementation would require all applications to output data in a format that the hardware desires – in this case, each of them would output to a 5-channel LPCM format (via codecs), and these then get merged together. This is shown in figure 3:

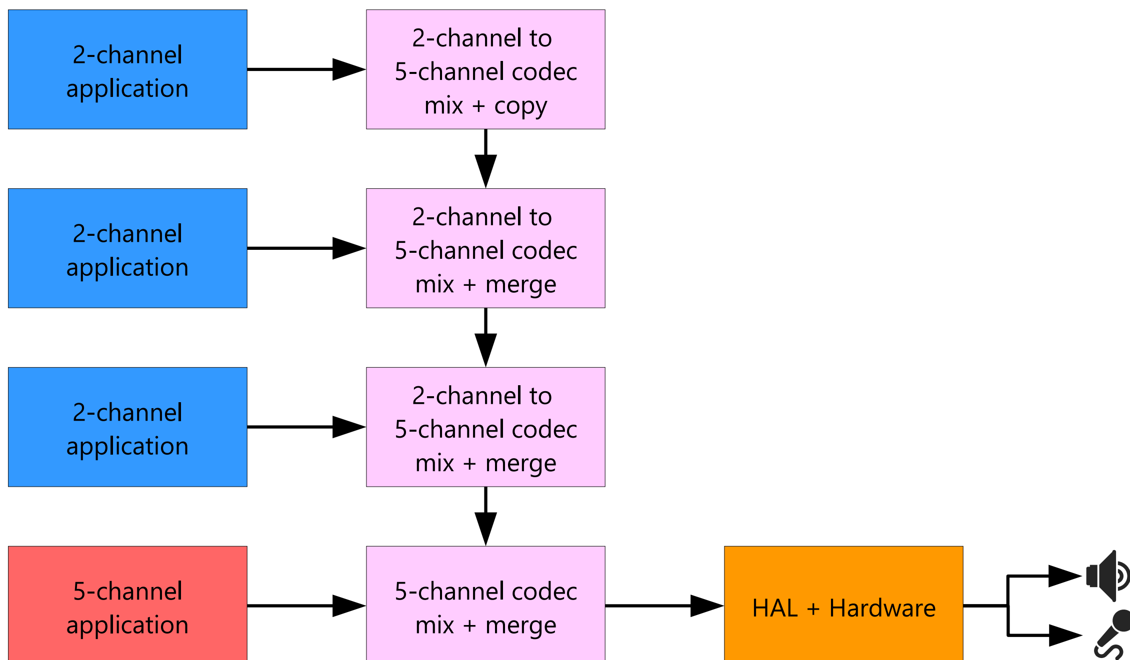

Figure 3: Simplified arrangement

Note that in all of these examples, it is assumed that the sample rate is the same for all applications.

How hardware uses a codec

If a piece of hardware requires a codec, it will ask the AudioManager to provide it for them. They will then pass the input data from the applications to the codec – which only needs to encode the data, and copy it to another area.

For example, Bluetooth outputting to an SBC formatted stream is shown in figure 4:

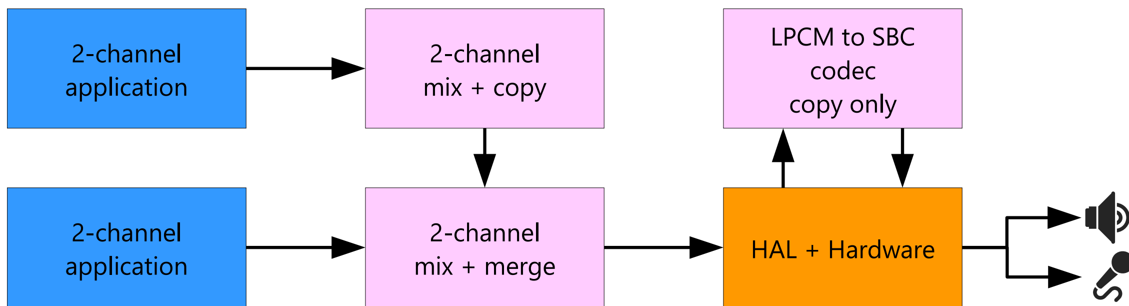

Figure 4: Bluetooth encoding

Native formats

Supporting native sample formats from an application to the hardware would require careful consideration. For example, if there is a single application that is providing SBC formatted data, then it can, in theory, pass unmolested through to a Bluetooth receiver (assuming no mixing is required), as shown in figure 6:

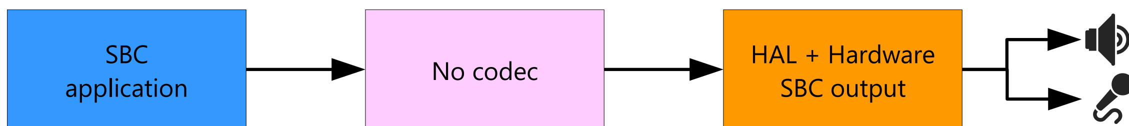

Figure 6: SBC only scenario

The problem comes when a second application wants to add its output to the stream – and it only supports LPCM (or another non-SBC formatted data stream).

In this case, the SBC stream must be converted into a format that allows merging – namely LPCM, and then the hardware will need to convert to SBC via the codec, as shown in figure 7:

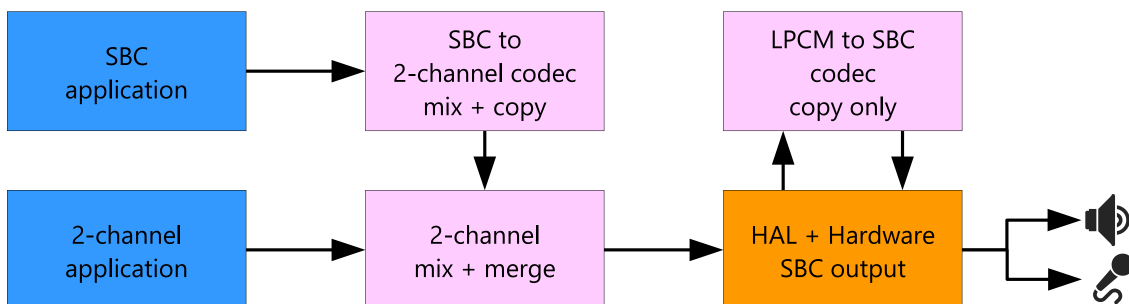

Figure 7: Merging SBC with LPCM

Note that if the hardware's SBC supported parameters does not match the application's output parameters, then the data will need to be recoded anyway, as shown in figure 8:

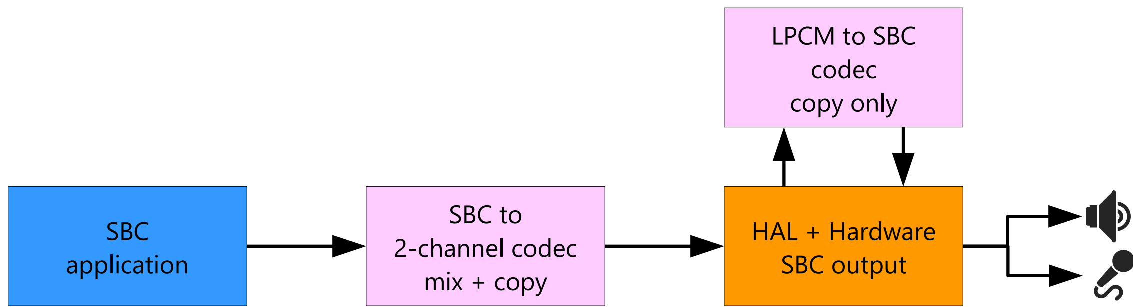

Figure 8: SBC recoding

Note that while this section mentions Bluetooth and SBC, the principal is the same for any encoding scheme.

When new applications join native streams (or leaves)

When a new application wants to join a native stream, three things are going to need to occur:

1. The existing application will need to get the codecs necessary to decode the native stream
2. The new application will need to get the right codec
3. The HAL will need to get the codecs necessary to encode the native stream

A service call can be issued to do this – and the sound paused until every application and hardware has caught up with the change (this implies the sample format is passed through as part of merging the audio).

Is it worth allowing native formats to go through without change?

At this point in time, it is worth asking this question; the amount of effort required to allow native formats will be quite a lot.

The benefits in not having to recode the data are:

- Audio format is retained, and no new losses are occurred in the decoding/encoding mechanisms
- Reduced CPU load
- Having to decide what the intermediary format is (mainly the sample rate)

The advantages in recoding are:

- Everything is performing LPCM mixing and merge/copy operations
- Simplified code structure
- Applications can join and leave at will

Since there are no native formats currently supported, it might be worth answering this question when they can be – however, the API should permit this at an early stage.

How this affects audio capture

Sound input is a lot simpler, since there are no mixer concerns. Applications that support the native format output from the hardware can simply receive the stream as it’s received, as shown in figure :

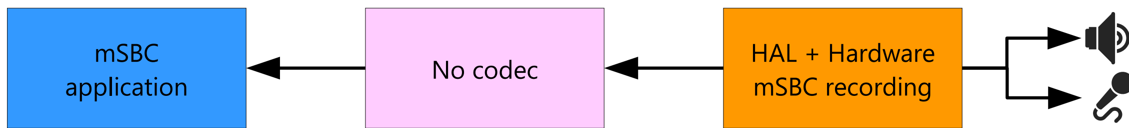

Figure 9: Simple audio capture

If an application requires a different format, then it requests a codec to take it to the format it requires, as shown in figure :

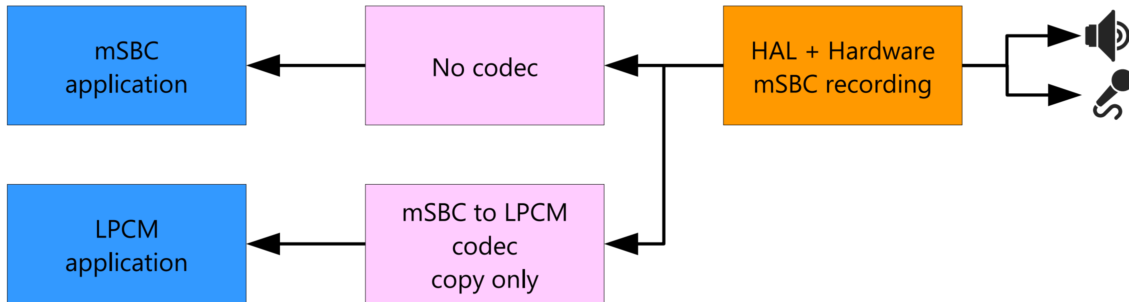

Figure 10: Audio capture from two applications

If more than one application needs the same output format, then you end up with an inefficiency – the codec can be created twice, and the decoding is performed twice:

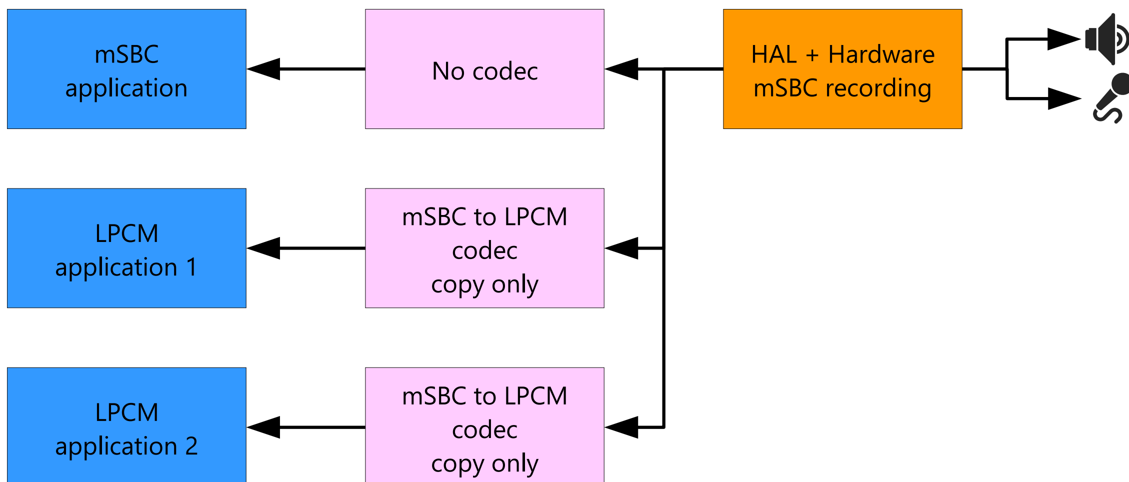

Figure 11: Unoptimised audio capture from three applications

It would be better if the codec is called once, and both applications can use the same decoded stream:

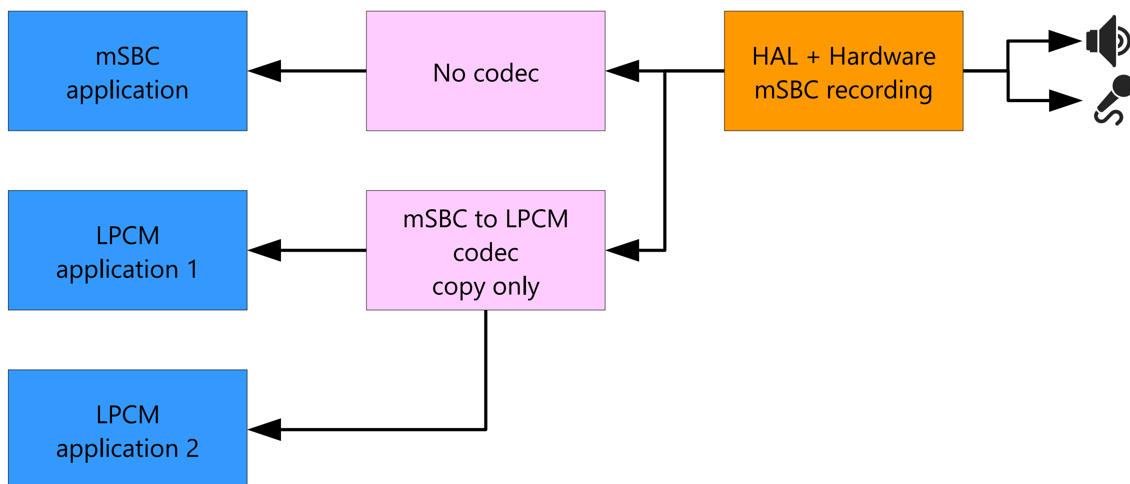

Figure 12: Optimised audio capture from three applications

The best way for this to occur is that applications register to record sound input and at the same time inform AudioManager which format they'd like to use as an output. AudioManager can then look at the list of applications requiring data, and group them together.

Audio capture limitations

In order to simplify things, it is recommended that all audio capture provided to applications at a fixed sample rate – which is the native one for the hardware. If an application needs a different sample rate, then it should create a codec and change the sample rate accordingly. AudioManager is only responsible for creating capture codecs that change the fundamental format.

In addition, audio capture will be buffered according to the hardware. It will indicate that there is data to be read, and applications need to deal with the amount of data present themselves. AudioManager will not buffer data into a temporary area for applications.

This means that if hardware sends data at a non-fixed rate (for example, 1024 samples for the first buffer, 512 samples for the second buffer, 1024 for the third, and 1536 for the fourth), then the application needs to handle the aperiodicity of the data.

How an application creates a codec

With these parameters in place, a codec is created with five pieces of information:

1. Input sample format
2. Input sample rate
3. Output sample format
4. Output sample rate
5. Mixer control, merge or copy required

Common codecs

Some codecs should exist in the system by default:

- 2-channel 16-bit RISC OS LPCM format to 2-channel 32-bit (AudioManager) LPCM format (with sample rate conversion, mixer control and merging/copying) – this is used by the SoundDMA shim.

- 2-channel 32-bit LPCM format to 2-channel 32-bit LPCM format (with sample rate conversion, mixer control and merging/copying) – this could be used by applications, and chained with some minimal codecs.

Additional codecs

With multi-channel devices, some more common codecs can be added – this would be done at the time of adding hardware support (i.e. won't be out of the box for the first version)

- 2-channel 32-bit AudioManager LPCM format to 5-channel 32-bit LPCM format (with sample rate conversion, mixer control and merging/copying).

Minimal codecs

Some minimal codecs could be created that would require additional support from the common codecs:

- MP3 to 2-channel 32-bit AudioManager LPCM format (no sample rate conversion, no mixer control, and no merging/copying).

Buffer manager: Getting sound into and out of hardware

All audio playback and recording is essentially the same. The hardware is yearning for data to play, and wants to give the data it has recorded.

It had always been my intention that the hardware interface would provide a signal to the sound system to say that new data was required (or available). How this occurs would depend on the hardware.

For I^2S devices, the interrupt would be generated by the hardware when the buffer was empty – for optimum performance, this would actually be the DMA hardware, rather than the I^2S hardware, although the I^2S hardware could be used without DMA when ultra-low latencies are required.

For USB devices, the interrupt would be generated by the USB driver, ostensibly under the control of the USB device.

For Bluetooth devices, this would be generated by the Bluetooth driver – however, this will be quite a way into the future.

Discovery interface: device identification

It has been suggested that a similar scheme to USB be adopted for identifying the capabilities of devices.

This is ostensibly a good idea as USB has to be able to cope with a multitude of devices, and software needs to be able to identify what it can do with each of them.

USB device descriptors

For those who have not read [USB002], the USB audio class definition defines a number of concepts that an audio system would have – the most interesting ones (for this document) being the input terminal (not necessarily recording), output terminal (not necessarily playback), mixer, and selector (and the first two are the most important for RISC OS sound).

‘Concepts’ are ‘chained’ together in order for sounds to flow through to the output audio channels. For example, Appendix B shows an example of a desktop microphone that consists of an input terminal (connected to the microphone), and an output terminal. The output terminal is connected to a USB input ‘stream’ (endpoint). For a speaker system, the input terminal would be the USB playback stream, and the output terminal would be the speakers.

Appendix C shows a more complex diagram consisting of several input terminals (one of which is a USB playback stream), several switches (to move audio between the terminals), and several output terminals (one of which is a USB recording stream).

Note that while an input terminal descriptor defines the number of channels, the output terminal descriptor does not – the host has to do a bit of work to identify this by looking at what the output terminal is attached to along the chain.

Control interface: changing settings

This is basically the next level for SoundControl, which would shim into this interface. It would allow access to various configuration settings within the hardware – for example, the mixer and selector concepts in USB.

Currently, there are not a lot of things that can be controlled by the SoundControl interface (the mixer), but more can be offered depending on the capabilities of the hardware being used.

This could also follow the USB device descriptor when applications need to identify what is available in the hardware.

Issues

There are a number of issues that complicate any audio system, and here, I would like to at least mention them – although not necessarily offer a solution.

Multi-format devices

If a device is able to accept or produce audio in multiple formats (for example, from a 2-channel to a 5-channel device), then this causes complications when applications want to reconfigure it.

This is also true (to a lesser extent) when the sample rate needs to be changed.

Therefore a system needs to be in place where an application can request the change in format. Other applications can veto this change (for example if they are playing a 5-channel stream on a 5-channel system, and a 2-channel player wants to select 2-channel only).

This would probably be best done by the application’s buffer fill code, since the change should be as quickly as possible.

If agreed, all applications will need to prepare for the new format by selecting new codecs (or changing the way they work).

Is this worth it?

It’s a lot of hassle to have to go through this – so I am wondering if we should only support the “best” format for multi-format devices.

Replicating streams on two devices

In some cases, it would be advantageous for sound to be able to be heard on two devices at the same time.

For example, if a user has some Bluetooth headphones, and an incoming Internet telephone call is received, then having both the Bluetooth headphones and the on-board speaker playing the ringing sound means if the user is not listening to the headphones, they are still aware of the incoming call.

If both devices support exactly the same sample rate and sample format, then the audio application providing the stream could pass its buffered data to both hardware streams at the same time.

However, if they are different in any way, then a translation would be needed (which the codec interface would be able to support).

Is it worth it?

I would probably argue that putting the same stream onto two devices would be more difficult than getting applications to play the streams on the two devices as individual streams. The occasions when both hardware requires the same stream would probably be few.

Likewise, recording from multiple devices at the same time would be merged by the applications, rather than the sound system. This can be assisted by codecs provided by the codec interface.

Changing audio sample rates

As mentioned in the "multi-format devices" section above, changing the sample rate would require all of the applications currently providing streams to the hardware to be aware of the sample rate change.

But which application should have precedence over any other when selecting the sample rate?

I would recommend that applications do not change the sample rate, unless the user has asked them to – in fact, it may even be necessary to make it a configuration setting.

However, if multiple applications want to set the sample rate, then the highest sample rate is the one that is chosen. Applications are informed of a proposed sample rate change via the buffer fill handlers, so they can reselect their codecs (although some codecs may be able to support a variable sample rate).

Note that sound may need to be paused while this happens – although if it is very quick, then it might not be noticeable (the hardware would need to change in synchrony).

I would also recommend that applications are not allowed to block sample rate changes. However, a system where a user can be alerted that the sample rate is about to change could be implemented.

Changing the default output device

If a user wants to change the default output device, then any applications that are not tied to a specific interface will need to probably reselect their codecs. They may also need to respect any change in sample rate.

Applications that are tied to the specific interface that is about to become a default one may need to be aware that other applications are joining the buffer fill. This could be done via the same mechanism.

Bibliography

USB001: USB organisation, USB Audio Data Formats, 1998, https://www.usb.org/sites/default/files/frmts10.pdf

USB002: USB org, USB Device Class Definition for Audio Devices, 1998, https://www.usb.org/sites/default/files/audio10.pdf

VAL001: ValdikSS, Audio over Bluetooth: most detailed information about profiles, codecs, and devices, 2019, https://habr.com/en/post/456182/