Report ITU-R BS.2388-5

(11/2024)

BS Series: Broadcasting service (sound)

Usage guidelines for the Audio Definition Model and Multichannel Audio Files

Foreword

The role of the Radiocommunication Sector is to ensure the rational, equitable, efficient and economical use of the radio-frequency spectrum by all radiocommunication services, including satellite services, and carry out studies without limit of frequency range on the basis of which Recommendations are adopted.

The regulatory and policy functions of the Radiocommunication Sector are performed by World and Regional Radiocommunication Conferences and Radiocommunication Assemblies supported by Study Groups.

Policy on Intellectual Property Right (IPR)

ITU-R policy on IPR is described in the Common Patent Policy for ITU-T/ITU-R/ISO/IEC referenced in Resolution ITU-R 1. Forms to be used for the submission of patent statements and licensing declarations by patent holders are available from https://www.itu.int/ITU-R/go/patents/en where the Guidelines for Implementation of the Common Patent Policy for ITU-T/ITU-R/ISO/IEC and the ITU-R patent information database can also be found.

	Series of ITU-R Reports
	(Also available online at https://www.itu.int/publ/R-REP/en)

Series	Title
BO	Satellite delivery
BR	Recording for production, archival and play-out; film for television
BS	**Broadcasting service (sound)**
BT	Broadcasting service (television)
F	Fixed service
M	Mobile, radiodetermination, amateur and related satellite services
P	Radiowave propagation
RA	Radio astronomy
RS	Remote sensing systems
S	Fixed-satellite service
SA	Space applications and meteorology
SF	Frequency sharing and coordination between fixed-satellite and fixed service systems
SM	Spectrum management
TF	Time signals and frequency standards emissions

Note: This ITU-R Report was approved in English by the Study Group under the procedure detailed in Resolution ITU-R 1.

Electronic Publication
Geneva, 2025

© ITU 2025

REPORT ITU-R BS.2388-5

Usage guidelines for the Audio Definition Model and Multichannel Audio Files

(2015-03/2017-10/2017-2018-2022-2024)

TABLE OF CONTENTS

1   Introduction

Recommendation ITU-R BS.2076 – The Audio Definition Model, is an open common metadata model for describing the technical format and content of audio files and streams. It primarily uses XML as its format language, and has been designed for incorporation into RIFF based audio files including those according to Recommendation ITU-R BS.2088 – Long-form file format for the international exchange of audio programme materials with metadata, on information technology media (BW64). The model can be converted to other languages, such as JSON, should the need arise; and also be used in conjunction with other file or stream formats.

This Report describes a set of typical use cases for the Audio Definition Model (ADM) and WAV-based files as well as recommended practices and commonly-used channel-based configurations. As the ADM is very flexible in how it can be used, it is possible to generate metadata that may prove difficult to interpret, therefore following the guidelines in this document will encourage consistent use by all.

As use of the ADM and BW64 increases more use cases and practices will appear, so this report should be kept up to date with any new requirements. The potential areas that may need guidelines are interoperation with streaming formats and renderers.

2   Use cases

Recommendation ITU-R BS.2076 lists a set of use cases that provides a general guide for the use of the model:

The use of the ADM is recommended especially for the following use-cases:

–   For applications requiring a generic metadata language for custom/proprietary formats (including codecs), or in the case where no metadata exists to describe what is needed.

–   For generating and parsing audio metadata with existing general-purpose tools.

–   Where experimental metadata can easily be added for an organisation's internal developments and where a human-readable and hand-editable file for describing audio configurations (such as describing a mixing studio channel configuration) in a consistent and translatable format is needed.

–   In WAV-based environments and workflows, where WAV-based broadcast applications wish to upgrade to be able to handle immersive content, while maintaining forward compatibility and handle legacy content.

–   For archiving of WAV-based content that also may include an extensive immersive metadata set.

As these use cases are quite general, a more specific and detailed set is required to enable a set of useful guidelines. The following sub-sections describe a set of typical practical use cases.

2.1   Generating BWF audio files from scratch

Possibly the simplest use case is generating BW64 files with ADM metadata from scratch.

This assumes that the audio that is being written to the file is known, so metadata is being generated with known information. This can be broken down into different use cases for different types of audio.

2.1.1   UC1.1: Common single group channel-based files

This is an audio file that consists of a common channel-based configuration, such as mono, stereo or 5.1. It is known what each channel is (e.g. Front Left, Front Right) and what audio it contains. It

only contains a single group of channels. This probably covers the vast majority of audio files that exist already.

2.1.2 UC1.2: Common multiple group channel-based files

The same as UC1.1, but the file will contain multiple groups of channels. For example, it may contain 4 stereo pairs or a 5.1 group and a stereo version. It is known what each channel is and what it contains.

2.1.3 UC1.3: Non-common channel-based files

This file contains channel-based audio, but the channels are not common definitions in common use. For example, they might be part of an experimental set of channel locations. It is known what each channel is.

2.1.4 UC1.4: Transformation/scene-based files

This file contains HOA (Higher Order Ambisonics) audio where all the components/channels are known.

2.1.5 UC1.5: Object-based files

This file contains object-based audio that can be either static or dynamic (i.e. changing its properties with time). The object properties are all known.

2.1.6 UC1.6: Mixed files

This file contains any combination of the channel, scene and object-based audio. A potentially popular combination would be a channel-based bed (typically stereo or 5.1) with object-based foreground objects overlaid.

2.2 Reading BWF audio files

The next obvious set of use cases is reading BWF files. It will be assumed that the file to be read contains correct ADM metadata, possibly generated from one of the UC1.x use cases.

2.2.1 UC2.1: Common single-group channel-based files

Reading a channel-based file containing a single group of common channels, such as stereo or 5.1 as described in UC1.1. The ADM metadata will be known and readable. The information from the ADM that could be of use will be the channel descriptions (such as intended speakers) and some content information such as loudness.

2.2.2 UC2.2: Common multiple group channel-based files

The same as UC2.1, but reading multiple groups as described in UC1.2. It will be important to ensure the groups of channels are treated correctly.

2.2.3 UC2.3: Non-common channel-based files

Reading non-common channel-based files such of those described in UC1.3. The useful information to read and interpret will be the channel positions, which could be unconventional.

2.2.4 UC2.4: Transformation/scene-based files

Reading scene-based (HOA) files as described in UC1.4. Interpreting each channel description as the correct HOA component to allow correct HOA decoding is important here.

2.2.5   UC2.5: Object-based files

Reading object-based files as described in UC1.5. Extracting the correct static and dynamic metadata and interpreting the timing is important here.

2.2.6   UC2.6: Mixed files

Reading mixed channel-based, scene-based, and object-based files as described in UC1.6. Ensuring all the audio is correctly extracted and matched with the correct metadata is one important factor.

2.3   Reading non-ADM WAV files

The majority of WAV files in existence do not contain any useful metadata to describe the format or content of the audio. If there is a need to read these files for conversion into BWF files with ADM metadata, then a decision needs to be made as to how to handle the lack of information. Following is a set of use cases covering this scenario.

2.3.1   UC3.1: One-, two-, five- and six-channel files

The vast majority of WAV files contain 1, 2, 5 or 6 channels that usually correspond to mono, stereo, 5.0 or 5.1 configurations. The number of channels in the file being read is known and they are likely to be channel-based, but there is no other information about the format or content.

2.3.2   UC3.2: Other numbers of channels

If presented with a 12-channel file there are a number of ways the configuration could be interpreted, such as 2x 5.1, or 6x stereo. There may be some information about the file to give clues, or it might be necessary to resort to examining the audio signals for guidance. This use case covers channel-based audio files with numbers of channels that do not provide an easy identification.

2.3.3   UC3.3: Multiple mono files

Sometimes multi-channel audio is stored as a collection of mono WAV files, with each file representing a particular channel. Without metadata in each file, the clues to determining the channel for each file could lie in the filename.

2.4   Generating BWF Files without information

Unlike use cases UC1.x where there is plenty of information about the format and content of the audio for the files being generated, this set of use cases covers the time when there is a lack of information and assumptions need to be made. This ties in closely with the use cases UC3.x where it is necessary to work out what is contained in the files being read.

2.4.1   UC4.1: Generating one-, two-, five- and six-channel files

Generating conventional channel-based configurations, possibly as read in the manner of UC3.1 or UC3.3. How to output enough useful ADM metadata to generate a useable BW64 file.

2.4.2   UC4.2: Generating other number of channels

Generating channel-based configurations, possibly read in the manner of UC3.2 or UC3.3. This will include determining whether multiple groups of audio may exist. There could be information about the format from configurations defined in documents such as Recommendation ITU-R BS.1738 or EBU R123.

3 Recommended practices

3.1 Using Common Definitions

Use cases: UC1.1, UC1.2, UC1.6, UC2.1, UC2.2, UC2.6, UC3.1, UC3.2, UC3.3, UC4.1, UC4.2.

To ensure consistent use of ADM definitions, and to save space and effort in defining them, a set of common ADM definitions covering a set of commonly used channel and pack definitions have been developed (Recommendation ITU-R BS.2094).

The Common Definitions are represented in an XML file that must be accessed in some way by any software that reads or writes BW64 files. These ADM definitions do not need to be carried in the audio file itself.

The elements covered in the Common Definitions are audioTrackFormat, audioStreamFormat, audioChannelFormat, and audioPackFormat. Currently they only cover commonly-used channel-based and scene-based definitions (though this may be extended to some matrix-based definitions in the future).

3.1.1 Using the Common Definitions when reading an audio file with an <axml> chunk

The steps when reading are:

1   Read in the Common Definitions XML file.

2   Read <chna> chunk from the audio file:

     a)   Inspect each row for the audioChannelFormatID or audioTrackFormatID reference.

         i)   If IDs occur in the Common Definitions, then use those channel definitions.

         ii)   If IDs do not occur in the Common Definitions, then refer to the <axml> chunk.

     b)   Inspect each row for the audioPackFormatID reference.

         iii)   If IDs occur in the Common Definitions, then use those pack definitions.

         iv)   If IDs do not occur in the Common Definitions, then refer to the <axml> chunk.

3   Read <axml> chunk for other ADM definitions:

     a)   Check for any references in the <axml> chunk to IDs in the Common Definitions.

3.1.2 Using the Common Definitions when writing an audio file with an <axml> chunk

The steps when writing are:

1   Read in the Common Definitions XML file.

2   Search the Common Definitions for the appropriate channel definitions:

     a)   If one exists, then use the ID and store for use in the <chna> chunk.

     b)   If it does not exist, then generate a custom channel description and add it to the <axml> chunk metadata, with the new ID ready for the <chna> chunk.

3   Search the Common Definitions for the appropriate pack definitions:

     a)   If it exists, then use the ID and store for the <chna> chunk.

     b)   If it does not exist, then generate a custom pack description and add it to the <axml> chunk metadata, with the new ID ready for the <chna> chunk.

4   Generate any other ADM metadata ready for the <axml> chunk.

     a)   Combine the chunks and audio for the output file.

3.2   Element IDs

Use cases: all

Each element in the ADM has its own identification attribute, such as audioChannelFormatID for audioChannelFormat. The use of these IDs is important as they are used for the elements to be able to reference each other, they are used by the <chna> chunk in the BW64 file, and are used to determine whether elements are common definitions or customs ones. The IDs uniquely identify each element, so care must be taken with their use.

3.2.1   ID prefixes

The IDs always have a prefix that corresponds to the element to which they belong. The prefix is followed by an underscore, then some hexadecimal digits. The table below shows the prefixes that should be used for each element.

Element	Prefix
audioProgramme	APR
audioContent	ACO
audioObject	AO
alternativeValueSet	AVS
audioPackFormat	AP
audioChannelFormat	AC
audioBlockFormat	AB
audioStreamFormat	AS
audioTrackFormat	AT
audioTrackUID	ATU

3.2.2   Hexadecimal codes

The format of the ID is a prefix followed by a number of hexadecimal digits. The hexadecimal letters (a to f) can be either upper or lower case, so any reading software must be case insensitive (so AC_0001001a is the same as AC_0001001A). The number and meaning of the digits depend upon the element used (see § 6 of Recommendation ITU-R BS.2076). Here are the recommended formats for each element:

3.2.2.1   audioProgramme

Format: `APR_wwww`

Hex digits `wwww`: Any value from `0001` to `FFFF`. This is used to identify the programme description. Values `0001` to `0FFF` are reserved for Common Definitions, while values `1000` to `FFFF` can be used for user defined or custom programmes.

A value of `1001` is often used for the first custom definitions in the XML, such as specified in Recommendation ITU-R BS.2168. The range of values for custom definitions may be further constrained if the composition is compliant with one or more ADM profile specifications.

Each audioProgrammeID within a file must be unique.

3.2.2.2   audioContent

Format: `ACO_wwww`

Hex digits `wwww`: Any value from `0001` to `FFFF`. Values `0001` to `0FFF` are reserved for Common Definitions, while values `1000` to `FFFF` can be used for user defined or custom contents.

A value of `1001` is often used for the first custom definitions in the XML, such as specified in Recommendation ITU-R BS.2168. The range of values for custom definitions may be further constrained if the composition is compliant with one or more ADM profile specifications.

Each audioContentID within a file must be unique.

3.2.2.3 audioObject

Format: `AO_wwww`

Hex digits `wwww`: Any value from `0001` to `FFFF`. A value from `1000` to `FFFF` for custom objects, which be any user defined objects; or a value from `0001` to `0FFF` are reserved for Common objects, which reside in an external Common Definitions file/resource (though no common objects currently exist but may in the future).

A value of `1001` is often used for the first custom definitions in the XML, such as specified in Recommendation ITU-R BS.2168. The range of values for custom definitions may be further constrained if the composition is compliant with one or more ADM profile specifications.

Each audioObjectID within a file must be unique.

3.2.2.4 alternativeValueSet

Format: `AVS_wwww_zzzz`

Hex digits `wwww`: These must match the digits of the audioObjectID of the parent audioObject element.

Hex digits `zzzz`: Any value from `0001` to `FFFF`. These often commence from "0001" for the first alternativeValueSet element within the parent audioObject and increase by 1 for each subsequent alternativeValueSet element within the parent audioObject in order of appearance in the XML, such as specified in § 2.2.1 of Recommendation ITU-R BS.2168.

Each alternativeValueSetID within a file must be unique.

3.2.2.5 audioPackFormat

Format: `AP_yyyyxxxx`

Hex digits `xxxx`: A value from `1000` to `FFFF` for custom packs; or a value from `0001` to `0FFF` for common packs, which reside in an external Common Definitions file/resource.

A value of `1001` is often used for the first custom definitions in the XML, such as specified in Recommendation ITU-R BS.2168. The range of values for custom definitions may be further constrained if the composition is compliant with one or more ADM profile specifications.

Hex digits `yyyy`: This value represents the type of audio contained in the pack, see Table 1 for common type values.

Each audioPackFormatID within a file must be unique.

3.2.2.6 audioChannelFormat

Format: `AC_yyyyxxxx`

Hex digits `xxxx`: A value from `1000` to `FFFF` for custom channels; or a value from `0001` to `0FFF` for common channels, which reside in an external Common Definitions file/resource.

A value of `1001` is often used for the first custom definitions in the XML, such as specified in Recommendation ITU-R BS.2168. The range of values for custom definitions may be further constrained if the composition is compliant with one or more ADM profile specifications.

This value should match the **audioStreamFormat** `xxxx` digits that references it.

Hex digits `yyyy`: This value represents the type of audio contained in the channel; see Table 1 for common type values.

Each audioChannelFormatID within a file must be unique.

3.2.2.7   audioBlockFormat

Format: `AB_yyyyxxxx_zzzzzzzz`

Hex digits `yyyyxxxx`: These must match the parent **audioChannelFormat** values.

Hex digits `zzzzzzzz`: This is a counter for the blocks in sequence within a channel The first block must be `00000001`, the second `00000002`, and so on (counting in hexadecimal).

3.2.2.8   audioStreamFormat

Format: `AS_yyyyxxxx`

Hex digits `xxxx`: A value from `1000` to `FFFF` for custom streams; or a value from `0001` to `0FFF` for common streams, which reside in an external Common Definitions file/resource. This value should match the **audioChannelFormat** `xxxx` digits to which the audioStreamFormat refers.

A value of `1001` is often used for the first custom definitions in the XML. The range of values for custom definitions may be further constrained if the composition is compliant with one or more ADM profile specifications.

Hex digits `yyyy`: This value represents the type of audio contained in the stream; see Table 1 for common type values.

Each audioStreamFormatID within a file must be unique.

For PCM audio, the audioStreamFormat and the audioTrackFormat should be omitted (see § 5.1 of Recommendation ITU-R BS.2076 for further details).

3.2.2.9   audioTrackFormat

Format: `AT_yyyyxxxx_zz`

Hex digits `xxxx`: A value from `1000` to `FFFF` for custom tracks; or a value from `0001` to `0FFF` for common tracks, which reside in an external Common Definitions file/resource.

A value of `1001` is often used for the first custom definitions in the XML. The range of values for custom definitions may be further constrained if the composition is compliant with one or more ADM profile specifications.

Hex digits `yyyy`: This value represents the type of audio contained in the track, see Table 1 for common type values.

Hex digits `zz`: This value represents the track number within a stream. This should start at `01` for the first track and increment for subsequent tracks.

The `yyyyxxxx` digits should match the **audioStreamFormat** `yyyyxxxx` digits to which the audioTrackFormat refers.

Each audioTrackFormatID within a file must be unique.

For PCM audio, the audioStreamFormat and the audioTrackFormat should be omitted (see § 5.1 of Recommendation ITU-R BS.2076 for further details).

3.2.2.10 audioTrackUID

Format: `ATU_vvvvvvvv`

Hex digits vvvvvvvv: A value from `00000001` to `FFFFFFFF` to uniquely identify an audio track (or part of one) within a file, such as specified in Recommendation ITU-R BS.2088, or when transported, such as specified in Recommendation ITU-R BS.2143. The value `00000000` must not be used for identifying this element as it is used to represent a silent track.

3.2.3 Recommended ID Numbering for Related Elements

For the audioTrackFormat, audioStreamFormat, audioChannelFormat and audioBlockFormat elements there is a very close relationship between them. Therefore, it is good practice to keep the IDs in connected elements well matched.

For PCM audio there is a one-to-one relationship between the essence and the resultant audioChannelFormat. Given this relationship, audioTrackFormat and audioStreamFormat are not required and are therefore not present in the ADM composition. Therefore, the recommended rules for the IDs are:

−   audioChannelFormatID: AC_YYYYXXXX   connects   with   audioPackFormatID AP_yyyyxxxx:

  •   YYYY = yyyy

  •   XXXX = 1??? (any value above 1000).

For coded audio the connection between audioTrackFormat and audioStreamFormat is an N-to-one relationship, and the other relationship is between audioStreamFormat and audioPackFormat (not audioChannelFormat as there are multiple channels in the stream). Therefore, the recommended rules for the IDs are:

−   audioStreamFormatID: AS_YYYYXXXX   connects   with   audioPackFormatID AP_yyyyxxxx:

  •   YYYY = yyyy

  •   XXXX = 1??? (any value above 1000).

−   audioTrackFormatID: AT_yyyyxxxx_nn   connects   with   audioStreamFormatID: AS_YYYYXXXX:

  •   yyyy = YYYY

  •   xxxx = XXXX

  •   nn = 01, 02, …

The audioBlockFormat element is the child of the audioChannelFormat element, so its ID follows this rule:

−   audioBlockFormatID: AB_YYYYXXXX_NNNNNNNN   with   the   parent audioChannelFormatID: AC_yyyyxxxx:

  •   YYYY = yyyy

  •   XXXX = xxxx

  •   NNNNNNNN = 00000001, 00000002, … (i.e. an incrementing counter for successive blocks within the channel).

3.3   Audio types

Use cases: all

The ADM is designed to cover any type of audio that needs describing. Currently, there are five categories:

1   Direct Speakers – commonly referred to as channel-based, such as stereo and 5.1.

2   Matrix – channels which don't feed directly to speakers, but which need combining via matrix operations such as Mid-Side and Lt/Rt.

3   Objects – object-based audio where channels of the audio include positional and other properties.

4   HOA – scene/transformation-based audio such as Ambisonics where channels represent spatial harmonic components.

5   Binaural – where the two channels are for the left and right ear.

There is nothing to stop more categories being used if required, but it is recommended that one of the five categories be used if at all possible.

The method of specifying which category is being used is done in three ways. The first is using the typeDefinition attribute, which uses a string identifier; the second is using the typeLabel attribute, which uses a numerical identifier; and the third is using the digits in the ID of the element. Either or both of typeDefinition and typeLabel must be used. The numerical part of the element's ID attribute (the yyyy digits) must match the type used.

Table 1 shows the strings and values used for each of the categories.

TABLE 1

Type definitions

Category	typeDefinition	typeLabel	ID hex digits (yyyy)
Direct Speakers	"DirectSpeakers"	"0001"	0001
Matrix	"Matrix"	"0002"	0002
Objects	"Objects"	"0003"	0003
HOA	"HOA"	"0004"	0004
Binaural	"Binaural"	"0005"	0005
User Custom	"User Custom"	"1000" to "FFFF"	1000 to FFFF

3.3.1   Format types

The audio can be stored in various ways in the file. The most common method in a WAV file is PCM; however, it is also possible to store audio in the tracks in other ways. To ensure the storage format is known, both audioStreamFormat and audioTrackFormat have format types associated with them to allow this format to be specified.

The method of specifying which category is being used is done in three ways. The first is using the formatDefinition attribute, which uses a string identifier; the second is using the formatLabel attribute, which uses a numerical identifier; and the third is using the digits in the ID of the element. Either or both of formatDefinition and formatLabel must be used. When the formatLabel is only used, the common definitions of the format types must be used. A value from 0001 to 0FFF is used for the common format types and a value from 1000 to FFFF is used for the custom format types.

Table 2 shows an example of the strings and values used for each of the categories.

TABLE 2

Example of format definitions

Format	formatDefinition	formatLabel
PCM	"PCM"	"0001"
MPEG-1*	"MPEG1"	"1002"
Dolby E*	"DolbyE"	"1003"
DTS*	"DTS"	"1004"

* MPEG-1, Dolby E and DTS are assumed as data_types of 4, 28 and 17 specified in SMPTE ST 338, respectively. The formatLabels for the custom format types are informative.

Missing both formatDefinition and formatLabel indicates PCM as the default format, so unknown data is assumed to be PCM unless otherwise specified.

Both Dolby E and DTS require two tracks to be used together to form one stream. Track 1 would be the 'left' track of the pair, and track 2 the 'right' track. The audioChannelFormat and audioPackFormats referred by the Dolby E and DTS audioStreamFormats are what the decoded audio would be (so typically 5.1).

3.4 <chna> chunk and IDs

Use cases: all

The <chna> chunk is the connection between the tracks in the WAV file and the ADM descriptions. Each line in the <chna> chunk corresponds to a track or part of a track. The ADM IDs that are contained in the chunk are audioTrackFormatID, audioPackFormatID and audioTrackUID. As audioTrackFormat always refers to an audioStreamFormat, which then refers to either an audioChannelFormat or an audioPackFormat; the IDs are enough to locate all the Format elements that describe the track. Recommendation ITU-R BS.2076-2 or later allows for audioTrackFormat and audioStreamFormat to be omitted in PCM only applications. When using this option, audioTrackFormat and audioStreamFormat elements are not present in the composition and the chna chunk contains references to the audioChannelformatID using the format AC_yyyyxxxx_00. The "00" suffix pads out the string to match the required number of bytes in the audioTrackFormatID field.

The audioTrackUIDs are used to uniquely identify the track, or segment of track, that has a description attached to it. These IDs are referred to from the audioObject element, which itself is referred from audioContent and audioProgramme, thus completing all the elements in the ADM.

3.4.1 Simple PCM channel-based files

For WAV files containing PCM channel-based audio (and scene-based too), the <chna> chunk is straightforward to use. Each line corresponds to each track in order along with its associated audioChannelFormatID and audioPackFormatID:

Track Number	audioTrackUID	audioChannelFormatID	audioPackFormatID
1	ATU_00000001	AC_00010001_00	AP_00010003
2	ATU_00000002	AC_00010002_00	AP_00010003
3	ATU_00000003	AC_00010003_00	AP_00010003
4	ATU_00000004	AC_00010004_00	AP_00010003
5	ATU_00000005	AC_00010005_00	AP_00010003
6	ATU_00000006	AC_00010006_00	AP_00010003

This example shows the chunk for a 5.1 file. Each track in the file contains a single audioTrackUID as each track only contains a single description (i.e. the channels exist for the complete duration of the file). The audioChannelFormatIDs refer to the definitions for PCM FrontLeft, FrontRight, etc.; and the audioPackFormatIDs all refer to the definition of the 5.1 pack.

Another thing to note with this example is all the Format IDs are using common definitions, i.e. their last four digits are 0FFF or less. As it only uses common definitions this means the WAV file's <axml> chunk does not need to carry any ADM metadata for these format definitions. The only elements the <axml> chunk may need is audioObject, audioContent, audioProgramme and audioTrackUID.

3.4.2   Simple matrix files

For WAV files containing both "DirectSpeakers" and "Matrix" audio, the <chna> chunk can be used to identify multiple audioTrackUIDs for one audio track. The audioPackFormatID field of particular track number and audioTrackUID in the <chna> chunk can reference an audioPackFormatID of the "Matrix" type. This "Matrix" type audioPackFormatID corresponds to an audioInputPackFormat.

Track Number	audioTrackUID	audioChannelFormatID	audioPackFormatID
1	ATU_00000001	AC_00010001_00	AP_00010003
2	ATU_00000002	AC_00010002_00	AP_00010003
3	ATU_00000003	AC_00010003_00	AP_00010003
4	ATU_00000004	AC_00010004_00	AP_00010003
5	ATU_00000005	AC_00010005_00	AP_00010003
6	ATU_00000006	AC_00010006_00	AP_00010003
1	ATU_00000011	AC_00020001_00	AP_00021003
2	ATU_00000012	AC_00020002_00	AP_00021003
3	ATU_00000013	AC_00020003_00	AP_00021003
4	ATU_00000014	AC_00020004_00	AP_00021003
5	ATU_00000015	AC_00020005_00	AP_00021003
6	ATU_00000016	AC_00020006_00	AP_00021003

The corresponding ADM XML is below:

```
<audioObject  audioObjectID="AO_1001"  audioObjectName="5.1"  start="00:00:00.00000"
duration="00:10:00.00000">
  <audioPackFormatIDRef>AP_00010003</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000001</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000002</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000003</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000004</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000005</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000006</audioTrackUIDRef>
</audioObject>

<audioObject  audioObjectID="AO_1002"  audioObjectName="5.1to2"  start="00:00:00.00000"
duration="00:10:00.00000">
  <audioPackFormatIDRef>AP_00021003</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000011</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000012</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000013</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000014</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000015</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000016</audioTrackUIDRef>
</audioObject>

<audioPackFormat audioPackFormatID="AP_00021003" typeDefinition="Matrix">
  <inputPackFormatIDRef>AP_00010003</inputPackFormatIDRef>
  <outputPackFormatIDRef>AP_00010002</outputPackFormatIDRef>
  <audioChannelFormatIDRef>AC_00021001</audioChannelFormatIDRef>
  <audioChannelFormatIDRef>AC_00021002</audioChannelFormatIDRef>
</audioPackFormat>

<audioChannelFormat audioChannelFormatID="AC_00021001" typeDefinition="Matrix">
  <audioBlockFormat audioBlockFormatID="AB_00021001_0000001">
    <outputChannelIDRef>AC_00010001</outputChannelIDRef>
     <matrix>
       <coefficient gain="1.0">AC_00010001</coefficient>
       <coefficient gain="0.7">AC_00010003</coefficient>
       <coefficient gain="1.0">AC_00010005</coefficient>
     </matrix>
  </audioBloackFormat>
</audioChannelFormat>
<audioChannelFormat audioChannelFormatID="AC_00021002" typeDefinition="Matrix">
  <audioBlockFormat audioBlockFormatID="AB_00021002_0000001">
    <outputChannelIDRef>AC_00010002</outputChannelIDRef>
     <matrix>
       <coefficient gain="1.0">AC_00010002</coefficient>
       <coefficient gain="0.7">AC_00010003</coefficient>
       <coefficient gain="1.0">AC_00010006</coefficient>
     </matrix>
  </audioBloackFormat>
</audioChannelFormat>
```

3.4.3　PCM object-based files

For WAV files containing PCM object-based audio, the <chna> chunk can be used to identify multiple objects in the same track if they don't overlap in time. An example <chna> chunk is shown here:

Track number	audioTrackUID	audioChannelFormatID	audioPackFormatID
1	ATU_00000001	AC_00031001_00	AP_00031001
2	ATU_00000002	AC_00031002_00	AP_00031002
2	ATU_00000003	AC_00031003_00	AP_00031003
2	ATU_00000004	AC_00031004_00	AP_00031004
3	ATU_00000005	AC_00031005_00	AP_00031005

In this example there are five audio objects, but as three of them do not overlap in time with each other they can share a single track (track 2 in this case) in the file. The excerpt of XML code below shows how the five audioObject elements are defined, with Obj2, Obj3 and Obj4 having non-overlapping time properties.

```
<audioObject  audioObjectID="AO_1001"  audioObjectName="Obj1"  start="00:00:00.00000"
duration="00:10:00.00000">
  <audioPackFormatIDRef>AP_00031001</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000001</audioTrackUIDRef>
</audioObject>

<audioObject  audioObjectID="AO_1002"  audioObjectName="Obj2"  start="00:02:00.00000"
duration="00:01:00.00000">
  <audioPackFormatIDRef>AP_00031002</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000002</audioTrackUIDRef>
</audioObject>

<audioObject  audioObjectID="AO_1003"  audioObjectName="Obj3"  start="00:04:00.00000"
duration="00:02:00.00000">
  <audioPackFormatIDRef>AP_00031003</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000003</audioTrackUIDRef>
</audioObject>

<audioObject  audioObjectID="AO_1004"  audioObjectName="Obj4"  start="00:07:00.00000"
duration="00:02:00.00000">
  <audioPackFormatIDRef>AP_00031004</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000004</audioTrackUIDRef>
</audioObject>

<audioObject  audioObjectID="AO_1005"  audioObjectName="Obj5"  start="00:00:00.00000"
duration="00:10:00.00000">
  <audioPackFormatIDRef>AP_00031005</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000005</audioTrackUIDRef>
</audioObject>
```

3.4.4   Coded audio files

While the majority of audio stored in the files will be PCM, there could be situations where coding audio could be stored. Coded audio could be stored across multiple tracks that need to be combined to produce a decodable bitstream. The ADM handles this using the audioStreamFormat with multiple audioTrackFormats connected to it. Often, coded audio contains multichannel audio in a single multi-track stream. An example of this is Dolby E carrying 5.1 channels in a 2-track stream. Where an audioStreamFormat element is describing multichannel stream it would refer to an audioPackFormat element as opposed to an audioChannelFormat element.

Therefore, the in <chna> chunk the audioPackFormatID reference does not need defining because it is already defined from audioTrackFormatID reference (via audioStreamFormat). A simple example of a <chna> chunk is shown below:

Track number	audioTrackUID	audioTrackFormatID	audioPackFormatID
1	ATU_00000001	AT_02011001_01	*
2	ATU_00000002	AT_02011001_02	*
* Fill with 11 zero value bytes.			

Note that the two audioTrackFormatIDs are identical apart from the two-digit suffix. Therefore, the audioStreamFormatID that these would refer to would be AS_02011001. The excerpt of XML below shows how these elements are defined to refer to the audioPackFormatID for a 5.1 pack.

```
<audioStreamFormat audioStreamFormatID="AS_02011001"
audioStreamFormatName="DolbyE_5.1"  formatLabel="1003" formatDefinition="DolbyE">
  <audioPackFormatIDRef>AP_00010003</audioPackFormatIDRef>
  <audioTrackFormatIDRef>AT_02011001_01</audioTrackFormatIDRef>
  <audioTrackFormatIDRef>AT_02011001_02</audioTrackFormatIDRef>
```

```
</audioStreamFormat>


<audioTrackFormat audioTrackFormatID="AT_02011001_01" audioTrackFormatName="DolbyE1"
formatLabel="1003" formatDefinition="DolbyE">
  <audioStreamFormatIDRef>AS_02011001</audioStreamFormatIDRef>
</audioTrackFormat>


<audioTrackFormat audioTrackFormatID="AT_02011001_02" audioTrackFormatName="DolbyE2"
formatLabel="1003" formatDefinition="DolbyE">
  <audioStreamFormatIDRef>AS_02011001</audioStreamFormatIDRef>
</audioTrackFormat>
```

3.5 Defaults for unknown audio inputs

Use cases: UC3.1, UC3.2, UC3.3

When reading traditional WAV files, with the intention of converting them into BW64 files, it is possible that no explicit information about the tracks will exist. Therefore, assumptions about the tracks and their order must be made. Clearly it makes sense to gather as much information about the input file as possible, to help identify the tracks; for example, if you received a 6-track file called "Effects_5.1.wav" then it is highly likely to be a 5.1 surround sound file.

Assuming no other knowledge about a file, apart from the number of tracks it contains, and that it contains a single programme or mix, then there are two approaches to take. The first approach to try uses the set of Common Definitions to match with the number of tracks in the file. The second approach to try is based on the Wave Format Extensible channel ordering. The two approaches are described in more detail in the following sub-sections.

3.5.1 Common Definitions approach

The set of Common Definitions contains both channel definitions and pack definitions. The pack definitions cover a range of commonly used channel-based configurations, and the number of channels in each of these packs can be used to match up with the number of tracks in the input WAV file. The method is simply this:

1   Read the number of tracks in the input file.

2   Find a pack in the Common Definitions with the same number of channels.

3   Generate a list of channel ID references from the chosen pack in the order given in the pack.

4   Match the stream ID references from the channel IDs and generate a <chna> chunk with the stream IDs and pack ID.

If stage 2 fails (i.e. there is no Common Definition pack of the correct size), then try the second approach below.

3.5.2 Wave Format Extensible approach

In some multichannel WAV files the method to handle the channel identification was to use the Wave Format Extensible extension, which provided a set of channel labels in a particular order. The set of channels in the Common Definitions have been given IDs in an order that matches the first 18 channel labels in Wave Format Extensible. The table below shows how the dwChannelMasks in Wave Format Extensible match the audioChannelFormat definitions in the Common Definitions. Some of the names differ slightly, and as Wave Format Extensible does not provide clear definitions of each of the channels, these were decided to be the closest matches.

Wave Format Extensible dwChannelMask		audioChannelFormat	
Speaker position	Flag bit	ID	Name
SPEAKER_FRONT_LEFT	0×1	AC_00010001	FrontLeft
SPEAKER_FRONT_RIGHT	0×2	AC_00010002	FrontRight
SPEAKER_FRONT_CENTER	0×4	AC_00010003	FrontCentre
SPEAKER_LOW_FREQUENCY	0×8	AC_00010004	LowFrequencyEffects
SPEAKER_BACK_LEFT	0×10	AC_00010005	SurroundLeft
SPEAKER_BACK_RIGHT	0×20	AC_00010006	SurroundRight
SPEAKER_FRONT_LEFT_OF_CENTER	0×40	AC_00010007	FrontLeftOfCentre
SPEAKER_FRONT_RIGHT_OF_CENTER	0×80	AC_00010008	FrontRightOfCentre
SPEAKER_BACK_CENTER	0×100	AC_00010009	BackCentre
SPEAKER_SIDE_LEFT	0×200	AC_0001000a	SideLeft
SPEAKER_SIDE_RIGHT	0×400	AC_0001000b	SideRight
SPEAKER_TOP_CENTER	0×800	AC_0001000c	TopCentre
SPEAKER_TOP_FRONT_LEFT	0×1000	AC_0001000d	TopFrontLeft
SPEAKER_TOP_FRONT_CENTER	0×2000	AC_0001000e	TopFrontCentre
SPEAKER_TOP_FRONT_RIGHT	0×4000	AC_0001000f	TopFrontRight
SPEAKER_TOP_BACK_LEFT	0×8000	AC_00010010	TopSurroundLeft
SPEAKER_TOP_BACK_CENTER	0×10000	AC_00010011	TopBackCentre
SPEAKER_TOP_BACK_RIGHT	0×20000	AC_00010012	TopSurroundRight

Therefore, this allows the approach for allocating IDs in the <chna> as simply placing them in numerical order as shown below:

Track Number	audioTrackUID	audioChannelFormatID	audioPackFormatID
1	ATU_00000001	AC_00010001_00	AP_00010001*
2	ATU_00000002	AC_00010002_00	AP_00010001*
3	ATU_00000003	AC_00010003_00	AP_00010001*
4	ATU_00000004	AC_00010004_00	AP_00010001*
NN	ATU_000000NN	AC_000100NN_00	AP_00010001*

* Depends on number of tracks.

The choice of audioPackFormatID is open to different options. One approach would be to generate a single custom pack containing the channels used, which will tie together the channels into a single pack. The other approach is to make each channel a mono pack (so use the ID AP_00010001), so each channel is treated independently. This second approach would be easier, and the recommended option to take and does not require any new definitions to be generated.

3.5.3   Generating other metadata for unknown audio inputs

The two approaches above make use of the Common Definitions to provide descriptions of the tracks for the audio file. There is therefore no need to generate definitions for the audioTrackFormat, audioStreamFormat, audioChannelFormat, and audioPackFormat elements (the format elements in other words). It would be acceptable to leave the <axml> chunk empty and just rely on the <chna> chunk, looking up the Common Definitions, which are external to the audio file. However, it is useful to consider generating at least an audioObject element that ties together the tracks in the file with

their format definitions more explicitly. This is particularly important if the audio file contains more than one mix of audio (for example, a 5.1 mix plus a stereo mix), as the different mixes need to be clearly identified.

Generation of the audioObjects is best explained with an example. Taking an 8-track audio file, where it has been identified that it contains a 5.1 (6 tracks) mix and a stereo (2 tracks) mix. Using the Common Definitions to determine the format elements, the <chna> chunk is as follows:

Track number	audioTrackUID	audioChannelFormatID	audioPackFormatID
1	ATU_00000001	AC_00010001_00	AP_00010003
2	ATU_00000002	AC_00010002_00	AP_00010003
3	ATU_00000003	AC_00010003_00	AP_00010003
4	ATU_00000004	AC_00010004_00	AP_00010003
5	ATU_00000005	AC_00010005_00	AP_00010003
6	ATU_00000006	AC_00010006_00	AP_00010003
7	ATU_00000007	AC_00010001_00	AP_00010002
8	ATU_00000008	AC_00010002_00	AP_00010002

As there are two mixes here then it is a good idea to generate two audioObjects to clearly identify them, rather than leaving it to assumptions about the nature of the packs. The following XML code (for inclusion in the <axml> chunk) shows how these objects could be generated:

```
<audioObject audioObjectID="AO_1001" audioObjectName="5.1_mix">
  <audioPackFormatIDRef>AP_00010003</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000001</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000002</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000003</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000004</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000005</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000006</audioTrackUIDRef>
</audioObject>

<audioObject audioObjectID="AO_1002" audioObjectName="stereo_mix">
  <audioPackFormatIDRef>AP_00010002</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000007</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000008</audioTrackUIDRef>
</audioObject>
```

Generating audioObjects is particularly important for files containing multiple mixes of the same pack type as it can become easy to misidentify how tracks are allocated.

3.5.4 Naming audio channels and loudspeaker labels for Cartesian coordinates

The audioChannelFormat has a name and loudspeaker label. Loudspeaker labels of playback sound systems for polar coordinates are specified in Recommendation ITU-R BS.2051. Loudspeaker labels can be expressed such as "M+030," "B-030" and "U+180," and consist of E+AAA or E-AAA. Here, E indicates layer names, they are "B" (Bottom layer, elevation angle -30 degrees), "M" (Middle layer, elevation angle 0 degree), "U" (Upper layer, elevation angle 30 degrees), "UH" (Upper height layer, elevation angle 45 degrees) and "T" (Top layer, elevation angle 90 degrees). AAA indicates azimuth angle, they are "+000" (azimuth angle 0 degree), "+030" (azimuth angle +30 degrees) and so on. Although loudspeaker labels for Cartesian coordinates are not specified in any ITU-R Texts, they can be used following the naming rule of loudspeaker labels for polar coordinates. It should also be noted that the Cartesian audioPackFormats defined in Recommendation ITU-R BS.2094 are intended to be played back through the loudspeakers defined by polar coordinates as specified Recommendation ITU-R BS.2051.

Z-axis values of "−1.00," "0.00" and "1.00" specify layers of "B" (Bottom layer), "M" (Middle layer), "U" (Upper layer), respectively. If a layer between the Bottom and Middle or Upper and Middle layers is required, then composite labels "BMz" or "UMz" are used (where z is replaced by a 2-digit position value representing the Z-axis). For example, "BM50" can be used when Z = −0.5 or "UM50" when Z = 0.5.

Y-axis values of "−1.00," "0.00" and "1.00" specify positions of "B" (Back), "Side" (Side) and "F" (Front), respectively. X-axis values of "−1.00," "0.00" and "1.00" specify positions of "L" (Left), "C" (Centre) and "R" (Right), respectively. The position of X = −1.00 and Y = 1.00 can be expressed by "FL" (Front Left).
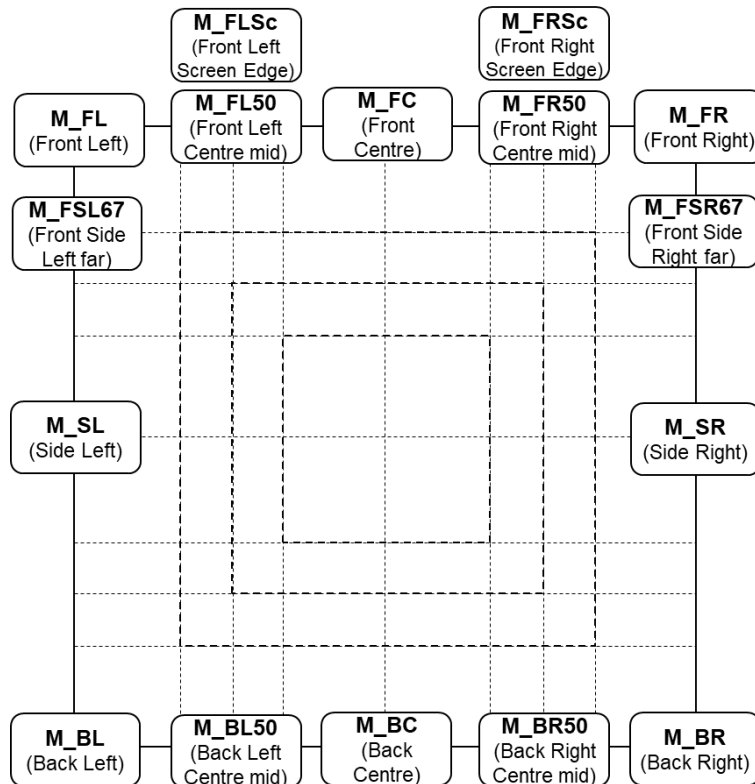
If a position between Front Left and Front Centre with Y = 1.00 is required, then composite label "FLx" is used (where x is replaced by a 2-digit position value representing the X-axis). For example, "FL50" can be used when X = −0.50.

If a position between Front Left and Side Left with X = −1.00 is required, then composite label "FSLy" is used (where y is replaced by a 2-digit position value representing the Y-axis). For example, "FSL50" can be used when Y = 0.50.
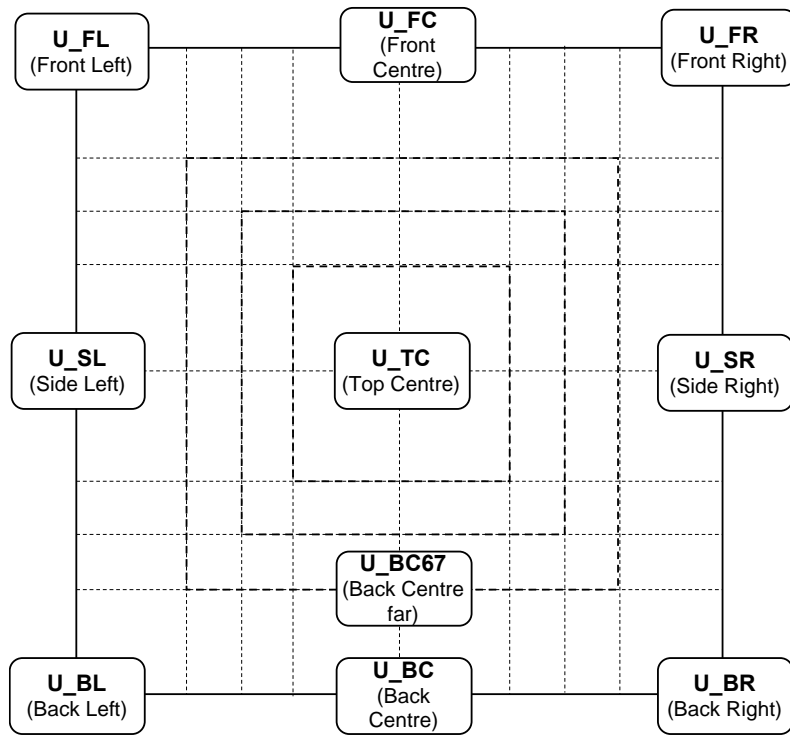
On the upper layer, "FLxy" (where x and y are replaced by 2-digit position values of X- and Y-axis) can be used for the Front Left area. If X = −0.333, Y = 0.667 and Z = 1.00, then composite label "U_FL3367" (Upper Front Left at x = −0.333 and y = 0.667) can be used. Following this naming rule, typical loudspeaker labels can be described as shown in Fig. 1.

FIGURE 1
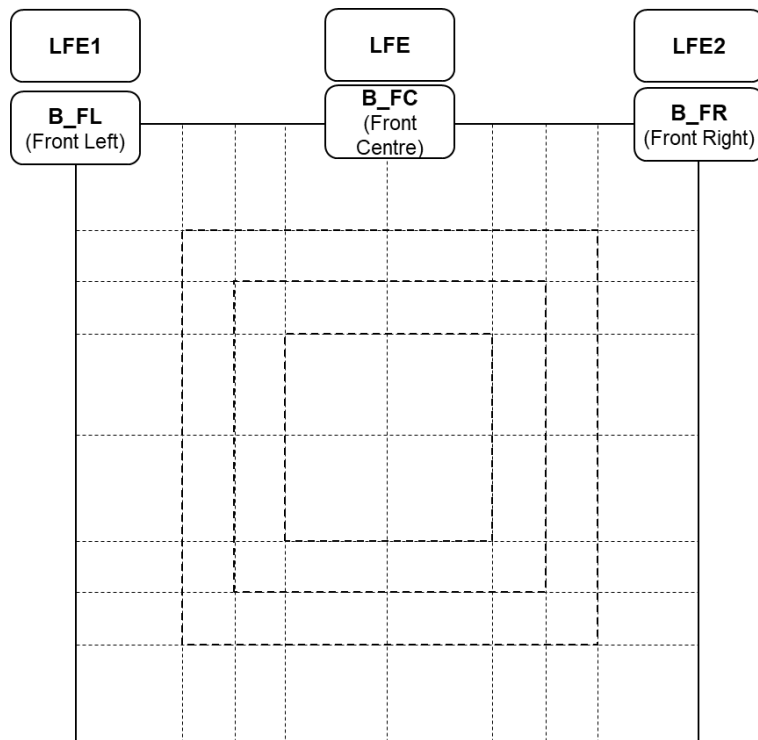
Loudspeaker labels for Cartesian coordinates
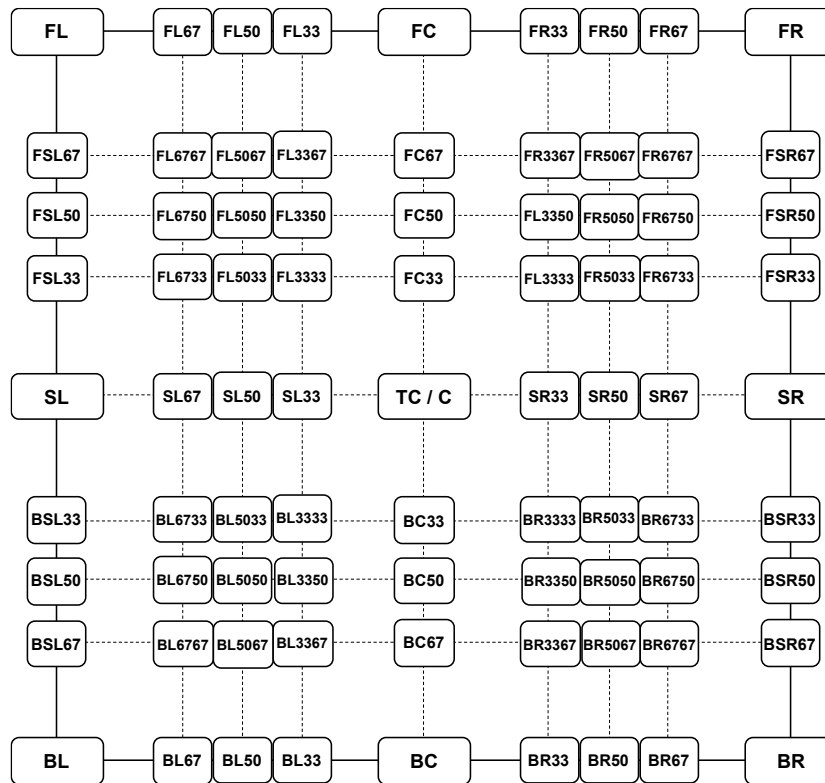
a) Middle layer

b) Upper layer

U_FL (Front Left)	U_FC (Front Centre)	U_FR (Front Right)
U_SL (Side Left)	U_TC (Top Centre)	U_SR (Side Right)
	U_BC67 (Back Centre far)	
U_BL (Back Left)	U_BC (Back Centre)	U_BR (Back Right)

c) Bottom (lower) layer

LFE1	LFE	LFE2
B_FL (Front Left)	B_FC (Front Centre)	B_FR (Front Right)

d) Typical loudspeaker labels for general layers

FL		FL67	FL50	FL33		FC		FR33	FR50	FR67		FR
FSL67		FL6767	FL5067	FL3367		FC67		FR3367	FR5067	FR6767		FSR67
FSL50		FL6750	FL5050	FL3350		FC50		FL3350	FR5050	FR6750		FSR50
FSL33		FL6733	FL5033	FL3333		FC33		FL3333	FR5033	FR6733		FSR33
SL		SL67	SL50	SL33		TC / C		SR33	SR50	SR67		SR
BSL33		BL6733	BL5033	BL3333		BC33		BR3333	BR5033	BR6733		BSR33
BSL50		BL6750	BL5050	BL3350		BC50		BR3350	BR5050	BR6750		BSR50
BSL67		BL6767	BL5067	BL3367		BC67		BR3367	BR5067	BR6767		BSR67
BL		BL67	BL50	BL33		BC		BR33	BR50	BR67		BR

e) Naming rules for Cartesian coordinated loudspeakers
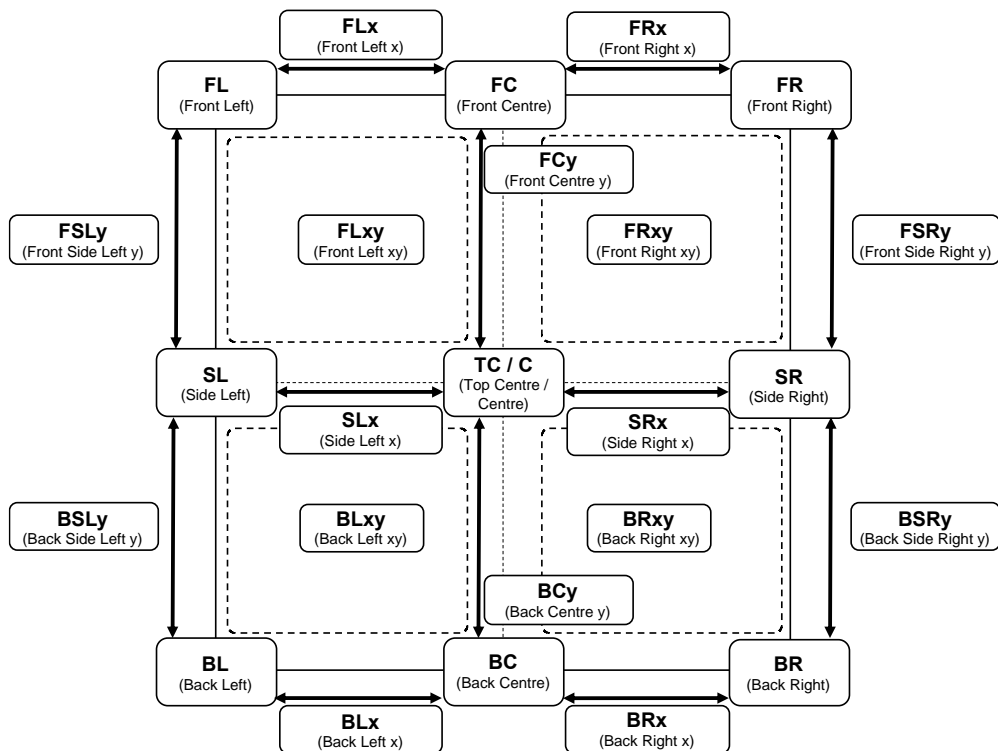
3.6 Times and durations

Use cases: UC1.5, UC2.5.

3.6.1 Timing attributes

The ADM has time and duration attributes in various elements, and their correct use is important to ensure things work correctly. The elements that contain time related parameters are:

| Element	Attribute	Meaning
audioProgramme	start	Time for the start of the programme
	end	Time for the end of the programme
audioObject	start	Start time of an object in seconds relative to the start of the programme.
	duration	Duration in seconds of an object.
audioBlockFormat	rtime	Start time in seconds of a block relative to the start of the object.
	duration	Duration in seconds of a block.

To help explain how these elements relate to each the other, the diagram in Fig. 2 shows the relationships.

The start of audioProgramme can be given a time (this is not essential, but recommended if it known), but this has no influence on the audio contents of the file; the audioProgramme is describing the whole file, so it starts at the first sample and ends at the last. In the example in Fig. 2, the time starts at 15:00:00:00 and ends at 15:30:00:00. It is important to ensure that *end – start* = the actual duration of the file. All the other timing takes the start of the audio file as time zero.

The audioObject *start* attribute corresponds to the start time of the object relative to the start of the file. The *duration* attribute corresponds to how long this object lasts. If this parameter is omitted the object lasts for the whole duration of the file. The audio samples in the file will correspond to the start and duration of the object as shown in the diagram (in the example this will be from 00:08:00.00 to 00:23:00.00).

The audioObject refers to an audioPackFormat (not shown for clarity) and an audioChannelFormat, which do not possess timing properties. The audioChannelFormat contains one or more audioBlockFormat that contain the finest level of timing information. The start (*rtime*) of the audioBlockFormat is relative to the start of the audioObject. So the positions of the audio samples in the file corresponding to a particular audioBlockFormat is found by adding its *rtime* to the *start* of the audioObject to get the first sample, and adding its *duration* to find the last sample.

FIGURE 2

Timing diagram

Timescale not to scale.
Seconds given to 2 d.p for clarity
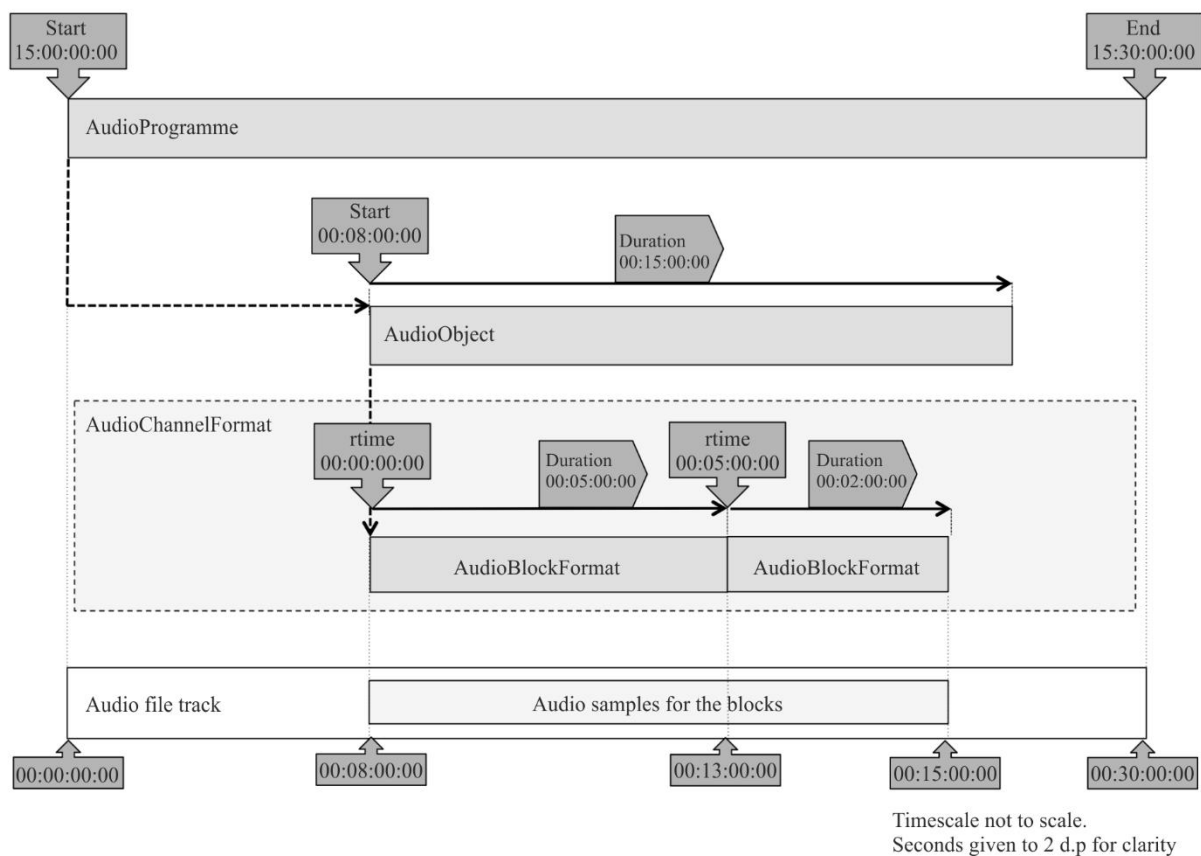
Report BS.2388-01

Is it possible to give timing values that are invalid or problematic, such as values that run off the end of the audio file or cause overlapping blocks. There are some rules that should be followed to ensure a correctly functioning file:

1   All the time and duration values must be non-negative.

2   The time attributes should be in the correct format:

    a)   HH:MM:SS:FF (where FF is the frame number) for the timecode attributes in audioProgramme. If there is any doubt about what the frame rate is then it is best to avoid using frame numbers and to use pure time as described in b) below.

    b)   HH:MM:SS.sssss (where sssss is at least 5 d.p. of the seconds) for the attributes in all elements other than audioProgramme. More than 5 d.p can be used, and is recommended for sampling rates greater than 48 kHz. For nanosecond precision 9 d.p. should be used.

3   The difference between the audioProgramme *start* and *end* time should match the duration of the audio file – within the precision of the timecode representation. Omit the *end* attribute if it uncertain what the length it.

4   The *start + duration* time of audioObjects should not be more than the duration of the file.

5   The *rtime + duration* time of audioBlockFormats should not be more than the *start + duration* of the referring audioObject if possible.

6   If the audioBlockFormats overrun the end of an audioObject, then it will be assumed the audio samples will only be read for the duration of the audioObject.
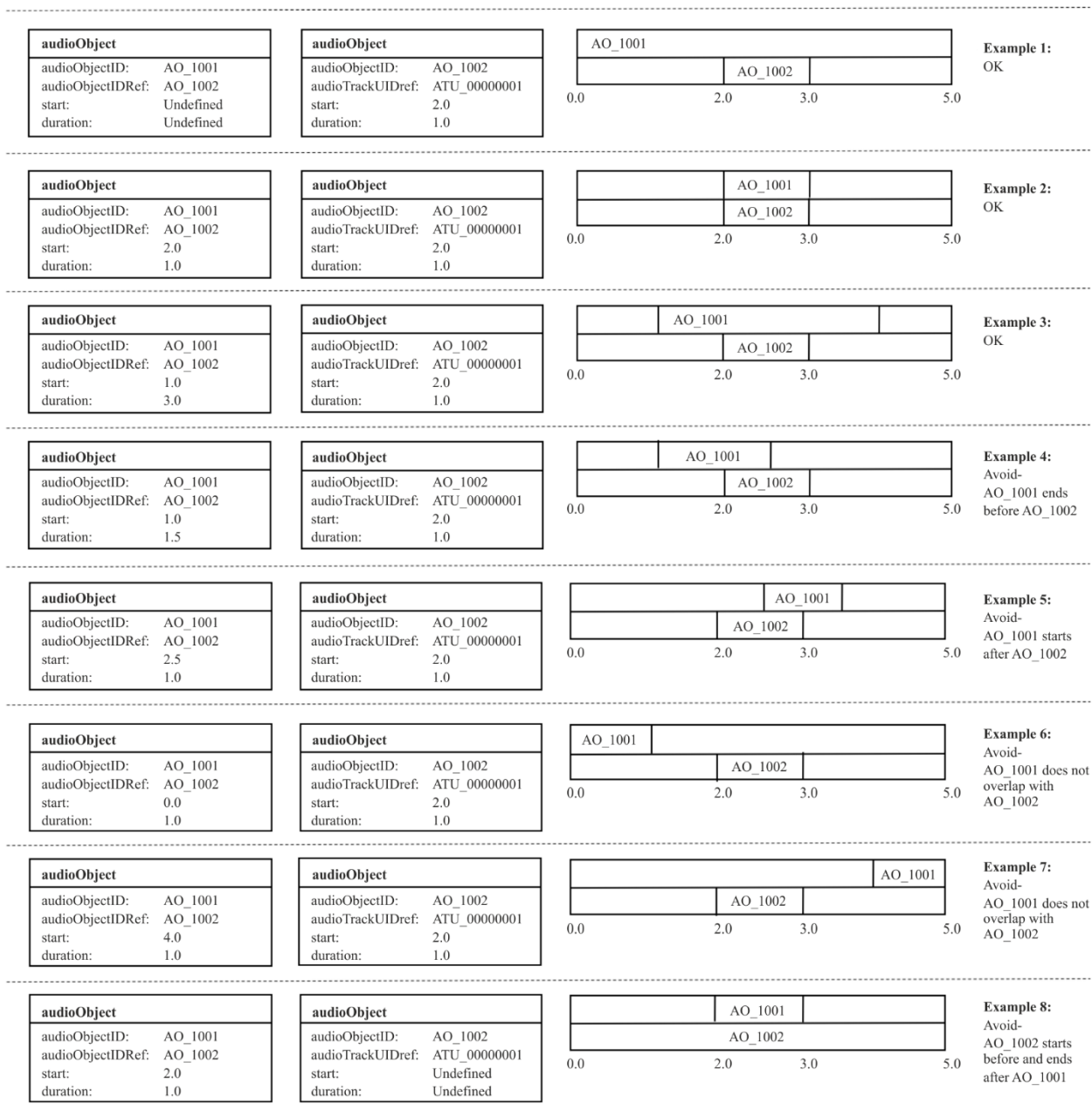
7   The order of audioBlockFormats within an audioChannelFormat should be chronological in the XML code.

8   Successive audioBlockFormats should be contiguous. Therefore the *rtime* of a block should equal the *rtime + duration* of the previous block.

9   It is recommended to have the first audioBlockFormat in an audioChannelFormat starting at 00:00:00.00000. Use the *start* time of the audioObject to set the starting time of a sequence of blocks.

3.6.2   Timing for Nested audioObjects

For many applications is it useful to use nested audioObjects. However, care must be taken to ensure that the timing of nested audioObjects don't cause them to become disconnected from each other. Figure 3 shows eight examples of how two connected audioObjects (AO_1001 refers to AO_1002) with different timing values can either work or fail.

It is also worth remembering that an audioObject's start time is relative to the start of the audioProgramme, not the start of the audioObject that refers to it.

FIGURE 3

Nested audioObjects timing examples

Example 1: OK

audioObject		audioObject	
audioObjectID:	AO_1001	audioObjectID:	AO_1002
audioObjectIDRef:	AO_1002	audioTrackUIDref:	ATU_00000001
start:	Undefined	start:	2.0
duration:	Undefined	duration:	1.0

AO_1001 (0.0 – 5.0), AO_1002 (2.0 – 3.0)

Example 2: OK

audioObject		audioObject	
audioObjectID:	AO_1001	audioObjectID:	AO_1002
audioObjectIDRef:	AO_1002	audioTrackUIDref:	ATU_00000001
start:	2.0	start:	2.0
duration:	1.0	duration:	1.0

AO_1001 (2.0 – 3.0), AO_1002 (2.0 – 3.0)

Example 3: OK

audioObject		audioObject	
audioObjectID:	AO_1001	audioObjectID:	AO_1002
audioObjectIDRef:	AO_1002	audioTrackUIDref:	ATU_00000001
start:	1.0	start:	2.0
duration:	3.0	duration:	1.0

AO_1001 (1.0 – 4.0), AO_1002 (2.0 – 3.0)

Example 4: Avoid – AO_1001 ends before AO_1002

audioObject		audioObject	
audioObjectID:	AO_1001	audioObjectID:	AO_1002
audioObjectIDRef:	AO_1002	audioTrackUIDref:	ATU_00000001
start:	1.0	start:	2.0
duration:	1.5	duration:	1.0

AO_1001 (1.0 – 2.5), AO_1002 (2.0 – 3.0)

Example 5: Avoid – AO_1001 starts after AO_1002

audioObject		audioObject	
audioObjectID:	AO_1001	audioObjectID:	AO_1002
audioObjectIDRef:	AO_1002	audioTrackUIDref:	ATU_00000001
start:	2.5	start:	2.0
duration:	1.0	duration:	1.0

AO_1001 (2.5 – 3.5), AO_1002 (2.0 – 3.0)

Example 6: Avoid – AO_1001 does not overlap with AO_1002

audioObject		audioObject	
audioObjectID:	AO_1001	audioObjectID:	AO_1002
audioObjectIDRef:	AO_1002	audioTrackUIDref:	ATU_00000001
start:	0.0	start:	2.0
duration:	1.0	duration:	1.0

AO_1001 (0.0 – 1.0), AO_1002 (2.0 – 3.0)

Example 7: Avoid – AO_1001 does not overlap with AO_1002

audioObject		audioObject	
audioObjectID:	AO_1001	audioObjectID:	AO_1002
audioObjectIDRef:	AO_1002	audioTrackUIDref:	ATU_00000001
start:	4.0	start:	2.0
duration:	1.0	duration:	1.0

AO_1001 (4.0 – 5.0), AO_1002 (2.0 – 3.0)

Example 8: Avoid – AO_1002 starts before and ends after AO_1001

audioObject		audioObject	
audioObjectID:	AO_1001	audioObjectID:	AO_1002
audioObjectIDRef:	AO_1002	audioTrackUIDref:	ATU_00000001
start:	2.0	start:	Undefined
duration:	1.0	duration:	Undefined

AO_1001 (2.0 – 3.0), AO_1002 (0.0 – 5.0)

Report BS.2388-02

3.6.3 Block sizes for dynamic objects

The audioBlockFormat element carries the parameters required for object-based audio with the time attributes to enable dynamic use. The audioChannelFormat can contain up to 4 294 967 295 audioBlockFormat blocks, and the duration of an audioBlockFormat can be 10 µs (i.e. less than a sample in length at a 48 kHz sample rate). It is therefore possible to use very fine temporal resolution, and thereby produce very large files. Such large files would be unwieldy, and there is unlikely to be the need for such short blocks, except when generating step changes in positional metadata. Specific ADM/S-ADM profiles may constrain block sizes, for example Recommendation ITU-R BS.2168. Otherwise, the following should be considered when deciding upon the sizes of the blocks:

1   If successive blocks carry identical parameter values, then combine them into one single block. In other words, new a block should only be generated when a change in parameter values is required.

2   Positions will be interpolated (unless chosen not to be), so smoothness of movement can be achieved without very fine blocks.

3   The speed and trajectory of the movement of objects. Slower objects could be represented with longer and fewer blocks.

4   Is there enough movement in an object to require multiple blocks? An object might not be moving by a noticeable amount, so one fixed position could suffice.

5   The larger the file, the more processing power and memory it will require. Too many small blocks will produce large files that will be slow to process and memory-hungry for digital audio workstations (DAWs).

6   Consider what will happen to the positional values if the coordinate system needs to be remapped. Will the interpolation between successive positions have a noticeably different trajectory when performed in another coordinate system? Keep the positional changes small enough to make such difference imperceptible.

The decisions depend upon the judgement of the sound engineer or software designer weighing up perceptual quality of the result and file size requirements. At this stage no recommended values are given, but some may be given in the future after more research.

3.6.4   Dealing with preambles

When programmes were exchanged using tapes rather than using files, it would be common practice for preambles and postambles to appear before and after the actual programme content. These would often include a video test card and line up tones, as well as a countdown timer, which are very useful when playing back from tape. While the need for these preambles are largely removed in a file-based world, there will still be programmes that have been transferred from tape to file that will contain them. Therefore, it makes sense to use the ADM timing metadata in a manner to ensure the file can be played out correctly.

Many broadcasters use an agreed start timecode for all their programmes, such as 10:00:00.00000. So anything that appears before this time is considered as a preamble and is not broadcast on playout. An object-based approach can be used to separate the preamble from the main programme by making them as separate objects. The preamble object can then start before the programme start time (so, before 10:00:00.000), with the main programme as a following object starting at 10:00:00.000. The upper diagram in Fig. 4 illustrates how this can be done. This method ensures the audioObjects all have non-negative start times. The lower diagram shows the wrong way to approach this, by starting the file at 10:00:00.000 and using a negative start time for the preamble audioObject.

FIGURE 4

Adding a preamble to a file
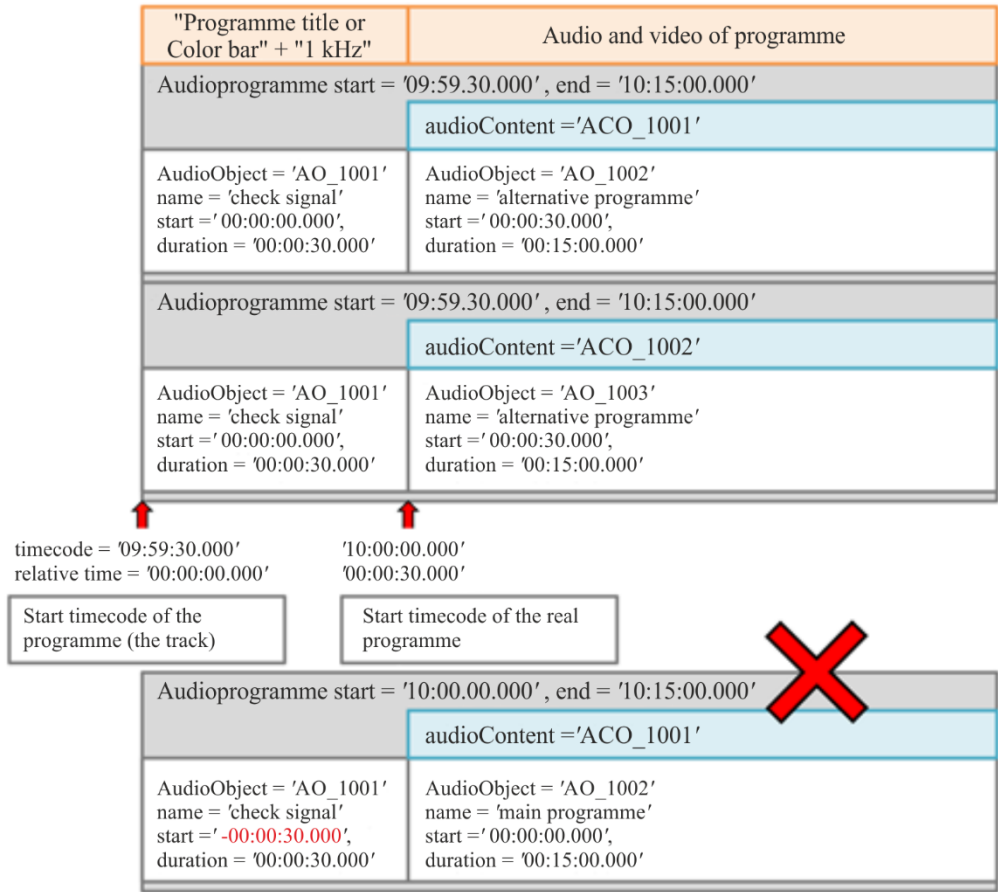
Report BS.2388-03

3.7   File management

While the BW64 format can handle files over 4GB in size, both for the audio samples and XML metadata, care must be taken when generating such large files. Section 3.6.3 describes how the block sizes with audioBlockFormat should be considered which have a large influence on the size of the <axml> chunk. When generating large files, it is worth bearing in mind the following:

1   The audio data (in the <data> chunk) may be directly read from file, rather the stored in memory in its entirety.

2   The XML metadata in the <axml> chunk is likely to be required to be read into memory in its entirety, even if it is later stored in a more compact form. Therefore having an <axml> chunk that is several gigabytes in size could cause memory problems.

3   As well as memory limitations, large files can also be slower to read and process.

4   The XML metadata also needs interpreting which could take a long time.

To overcome these issues there are some measures that can be taken:

1   Consider increasing audioBlockFormat blocks sizes based on the considerations described in § 3.6.3.

2   Avoid duplicating metadata. The "format" ADM elements are not directly tied to audio data, so make use of the Common Definitions if possible and identify any audioChannelFormat elements that are identical which can be reused.

3   Take advantage of non-overlapping objects. These can share audio tracks to reduce the number of tracks used in the file.

The advice given in this section is not an exhaustive list and should be added to or refined in the future if required.

3.8   \<fmt\> Chunk handling

The \<fmt\> chunk in WAV files is a mandatory that specifies the format of the data within the file. The \<fmt\> chunk is defined as follows:

```
<fmt> ->fmt(<common-fields>
           <format-specific-fields>)
<common-fields> ->
    Struct {
        WORD  wFormatTag;              // Format category
        WORD  nChannels;              // Number of channels
        DWORD nSamplesPerSec;         // Sampling rate
        DWORD nAvgBytesPerSec;        // For buffer estimation
        WORD  nBlockAlign;            // Data block size
    }

<format-specific-fields>) ->
    Struct {
        WORD  WBitsPerSample;         // Bits per sample
        WORD  cbSize;                 // Size of the extension (0 or 22)
        WORD wValidBitsPerSample;     // Number of valid bits
        DWORD dwChannelMask;          // Speaker position mask
        CHAR[16] SubFormat;           // GUID, including the data format code
    }
```

The \<format-specific-fields\> are optional and are used in the Wave Format Extensible mode. The format code in wFormatTag specifies the type of data used, and a small selection of these types are shown below:

wFormatTag	Symbol	Format
0×0000	WAVE_FORMAT_UNKNOWN	Unknown
0×0001	WAVE_FORMAT_PCM	PCM
0×0003	WAVE_FORMAT_IEEE_FLOAT	IEEE float
0×FFFE	WAVE_FORMAT_EXTENSIBLE	Determined by SubFormat

If the format is set to WAVE_FORMAT_EXTENSIBLE then the file uses the \<format-specific-fields\> with the dwChannelMask to specify the channels.

When reading a WAV file the \<format-specific-fields\> may or may not be specified, but if they do exist they should be read to try and determine as much about the channels and format as possible.

When generating a BW64 file, it is recommended to avoid using the Wave Format Extensible mode as there is the risk of generating information that contradicts the \<chna\> and \<axml\> chunk information. The recommended approach is to only use one of two wFormatTag values according to these rules:

–   If all the audio track in the file are PCM then set wFormatTag to 0x0001 (PCM)

–   If any of the audio tracks are non-PCM then set wFormatTag to 0x0000 (unknown).

3.9   Ensuring streaming compatibility

The way in which the ADM is used, and the way in which multichannel audio files are created, will have an impact on the efficiency with which the audio data can later be streamed. The use cases in this area are still being developed.

3.10   Interactivity and ensembles of audioObjects

AudioObjects may have an "interact" attribute, and audioObjectInteraction sub-elements. These signal that the user may interact with the audioObject, changing position or gain, or turning off an object. In addition, audioObjectInteraction sub-elements can be used to signal limits on the size of changes to position or gain.

Note that positions and gains are sub-elements not in audioObjects, but of audioBlockFormats. An audioBlockFormat represents a single sequence of audioChannelFormat samples within a specified time interval. An audioChannelFormat represents a single sequence of audio samples on which some action may be performed in a rendered scene. An audioChannelFormat is sub-divided in the time domain into one or more audioBlockFormats.

Because audioObjects may refer to other audioObjects, it is possible to interact with a referent object in order to change the referenced objects. It may also possible to interact with a referenced audioObject, without changing the referent object.

An example scenario in which this might be desired, and the expected behaviour, are described in §§ 3.10.1 and 3.10.2.

3.10.1   Example of interaction with an ensemble of audioObjects

A broadcast that includes a performance by a string trio uses multiple objects to create the sound of the orchestra. A different audioObject is used for the sound of each instrument. An audioObject named "ensemble" refers to audioObjects named "violin", "viola", and "cello".

The "violin", "viola", and "cello" audioObjects each refer to their own audioPackFormat, audioChannelFormat, and audioBlockFormat. Each audioBlockFormat includes a position from which the renderer should, by default, render the sound of the respective instrument. The positions signalled in the metadata correspond to a reasonable spacing and order of the instruments:

<div align="center">violin --- viola --- cello</div>

By default, the three instruments are spaced 30 degrees apart, symmetrically either side of 0 degrees.

The broadcaster allows the listener to change the azimuth of the ensemble by ±90 degrees. It does this by enabling interaction on the "ensemble" audioObject, and setting the positionInteractionRange minimum and maximum attributes accordingly.

The broadcaster further allows the listener to move the instruments within the ensemble to a limited extent. It does this by enabling interaction on the "violin" and "cello" audioObjects. The positionInteractionRange limits on the "violin" and "cello" are set such that the overall width of the ensemble may be reduced to 30 degrees.

The desired behaviour is that an interaction with a referent object is applied to all referenced objects. Interaction with a referenced object is not applied to the referent object.

Interaction with the "ensemble" audioObject, applying a change of azimuth of 45 degrees, causes a change in azimuth of each of the "violin", "viola", and "cello" audioBlockFormats.

A subsequent interaction with the "violin" and "cello" audioObjects that applies a change of azimuth of −10 and 10 degrees respectively is still considered to be within the interactionRange limit for those objects, because the earlier interaction with the referent "ensemble" object should not be taken into account.

In this way, the listener may change both the width and the position of the ensemble, with a tighter constraint on the change of width than on the change of position.
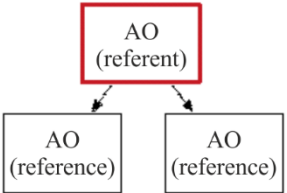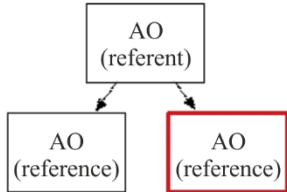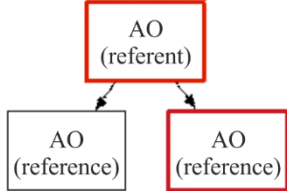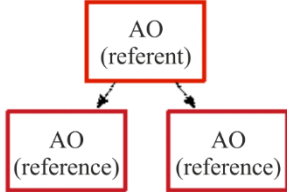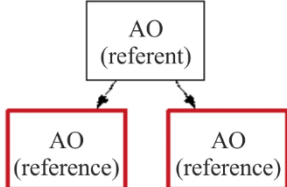
The control used by the listener to effect the interaction might be presented in a way that facilitates either rotation or translation. The signalling of interactionRange limits would logically be done using a coordinate system that is consistent with the control. For example, if a control were provided to change the azimuth of a sound source, it would be logical to signal the interactionRange in polar coordinates. Conversely, if a control were provided to move a sound source in a straight line from side to side, it would be logical to signal the range in Cartesian coordinates.

The value of attributes in the position or gain sub-elements of an audioBlockformat can change over time. The existing offsets caused by earlier interactions should persist after a change to the attributes.

There are several combinations of interaction being enabled or not in an ensemble of audioObjects. The following diagram in Fig.5 illustrates the possibilities and the interactions that are allowed as a result, using position as the example.

FIGURE 5

Possible combinations of positional interactivity in an ensemble of audioObjects, and the changes enabled

3.10.2 Behaviour of interaction with ensembles of audioObjects

If an audioObject allows interaction, the change to an attribute that can be set by the user should be within the limits of the interaction range of that audioObject. In this context, a "change" is the difference between a condition before and after the interaction.

The resultant position and gain of a sound source is the combination of the attributes of the position and gain sub-elements of the audioBlockFormat and all the changes caused by interaction in the hierarchy of audioObjects that refer to the audioBlockFormat.

3.11 Multiple audioProgrammes

It is possible to define more than one audioProgramme within a BW64 file, and there could be several different reasons for doing so. These include:

– Different language versions of the same programme;

– Different length versions of the same programme;

– Versions of the programme that have been mixed on different reproduction systems (e.g. a stereo version and a object-based version);

– Adjustments in content to cater for age ratings or music rights.

There could also be the need for carrying multiple unrelated programmes in a single file, but it is more likely that a file will contain versions of the same programme.

3.11.1 Using multiple audioProgrammes versus audioContents and audioObjects

Taking the example of a programme with two different language versions, this can be approached in two different ways. The first approach consists in two different audioProgrammes to be used, one for each language. The second approach is a single audioProgramme with interactive audioObjects that are complementary. Figure 6 shows the two audioProgramme approach with an English and German version, and both audioProgrammes share the same music and effects audioContent. Figure 7 shows the single audioProgramme version where either the English or German audioObject can be selected.
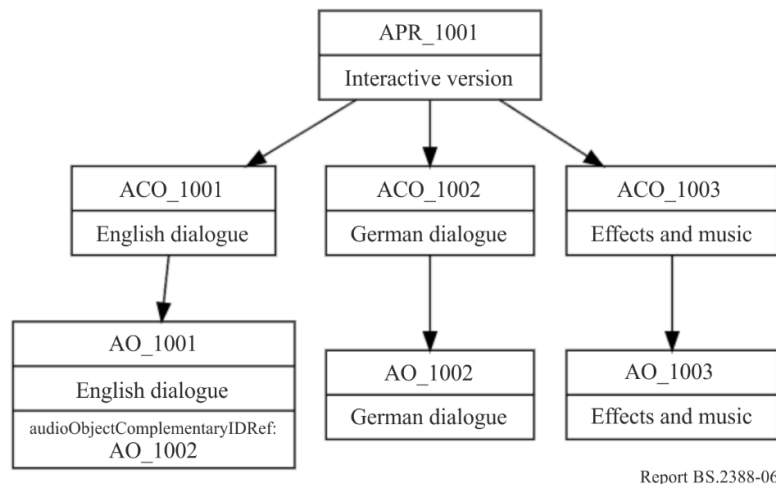
FIGURE 6

Two audioProgramme approach

Report BS.2388-05
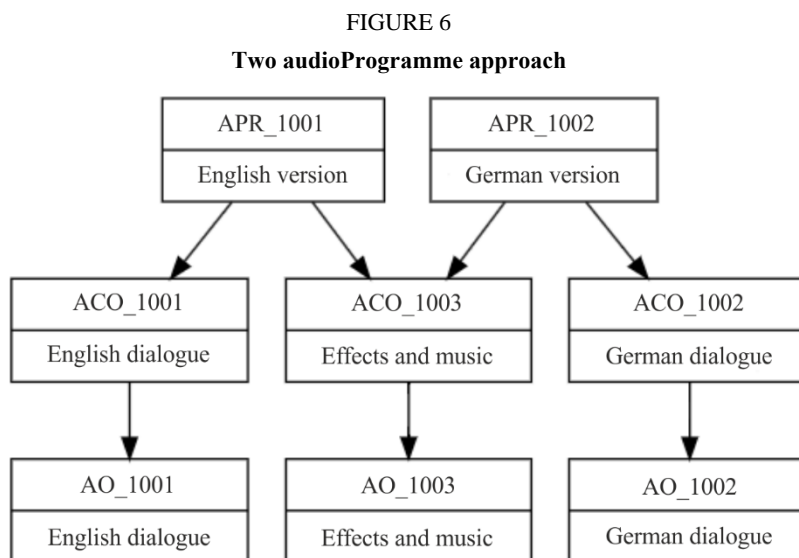
FIGURE 7

Single audioProgramme approach

Report BS.2388-06

Which approach should be used when?

The multiple audioProgramme approach should be used when interactivity is not expected to be used. In the example given, the playout system is expected to be fixed to either English or German and to select the appropriate programme to use. The user is not expected to switch between languages while watching.

The single audioProgramme approach should be used when interactivity is expected to be used. In the example, some language selectors are expected to be available. The user can therefore switch between languages while watching.

3.11.2   Default audioProgramme

When a file contains more than one audioProgramme, and only one audioProgramme can be handled by the receiver of the file, then a choice has to be made to select one. In the example of a multilanguage programme, the target language may be known. By inspecting the language parameters in audioProgramme (or the related audioContents) should be determined which audioProgramme to use.

However, there might be situations where no other information is known about which audioPrograame should be selected, yet one needs to be chosen to play or process. When there is no other information to help select an audioProgramme, then a default one must be chosen. The simplest rule for determining which is the default audioProgramme to use it to select the one with the lowest audioProgrammeID value. In the example shown in Fig. 6, the default audioProgramme would be APR_1001.

3.12   Using the 'importance' parameters

The 'importance' parameter appears in the following elements: audioObject, audioPackFormat and audioBlockFormat. The integer value of the parameter ranges from 0 (least important) to 10 (most important). The main use for the importance parameter is when compromises have to made to the quantity of audio objects in the file. For example, a renderer on a device can only handle 64 objects at once, but the ADM file it receives contains 90 objects, so some sort of reduction has to be made. A pre-processer can make this reduction from 90 to 64, but by having some sort of guide to which objects are more important than others it can make this reduction in a manner that retains the content creator's intent more closely.

The behaviour for the importance parameter differs between each of the elements it is used in.

3.12.1 The audioBlockFormat importance parameter

While the importance parameter is in the audioBlockFormat, it does not mean it should be treated as a dynamic parameter, as stated in Recommendation ITU-R BS.2076-1 § 9.2. Therefore, it should be considered to be a fixed value for that particular audioChannelFormat. While Recommendation ITU-R BS.2076 suggests that audioBlockFormats below a certain value are discarded, it is not recommended to take this action. The importance parameter in audioPackFormat should take precedent for this action, and the audioBlockFormat importance parameter should only be used for informative uses.

3.12.2 The audioPackFormat importance parameter

The importance parameter in audioPackFormat can be used to make compromises in the spatial quality of an audio object. As an audioPackFormat is intended to group together channels that belong together, they can be considered to be equally important. However, audioPackFormats can be nested, so less important packs can be referred to from a parent pack. Figure 8 shows a simple use of nested audioPackFormats, where a parent pack ('Speech') contains two child packs ('SpeechDirect') and ('SpeechEarly'). The 'SpeechDirect' pack contains a single channel and has an importance of 10, so should never get discarded. The 'SpeechEarly' pack contains two channels and has an importance of 5, so may get discarded. In this example, the 'SpeechDirect' could contain the direct signal of some dialogue, so it critical to the sound; but the 'SpeechEarly' could contain some reverb sounds of the dialogue, so discarding them will only adjust the feel of the sound slightly.

FIGURE 8

Nesting audioPackFormats for importance parameter use
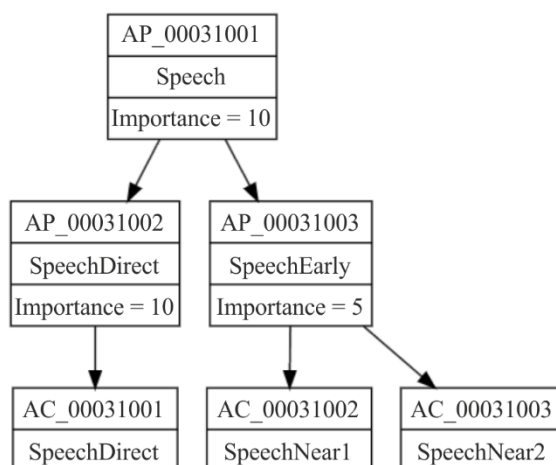
Report BS.2388-07

So, a pre-processor that aims to reduce the number of objects or tracks in a file, could discard or ignore the audioPackFormats below a certain importance value. The quality might be compromised, but the actual object will still exist. So in the example given, the 'Speech' pack will either contain one or three channels depending on the importance level chosen.

This is not restricted to the 'Objects' type either, and is just as useful for other types too. It is also worth remembering that audioPackFormats can be reused for different content, so any importance parameter will be reused for the different content in the same way. So the importance parameter here is very much for compromising the quality of the format, not the content.

3.12.3   The audioObject importance parameter

The importance parameter in audioObject has a slightly different behaviour from the of the audioPackFormat version. Whereas the audioPackFormat makes compromises on quality (primarily the spatial quality), the audioObject version allows less important sounds to be potentially discarded too. Unlike the audioPackFormat, this allows for content-dependant compromises to be made. For example, minor atmospheric effects could be discarded to ensure the main effects are properly reproduced.

Discarding is not the only option for handling audioObjects of low importance. A processor could also combine audioObjects that share similar properties, such as close positional proximity.

As with audioPackFormats, nesting can be used to allow selection of the audioObjects that can either be discarded or combined. The example shown in Fig. 9 shows and audioObject (AO_1001) that contains two sub-audioObjects, Speech (AO_1002) and RoomReverb (AO_1003). The RoomReverb audioObject has a lower importance, so it could be potentionally discarded, as it contains less importance sounds. Alternatively, if the processor is so designed, it could examine the properties of AO_1002 and AO_1003 and merge them together to make a single combined object (this will have to include mixing of the audio samples too).

FIGURE 9

Nesting audioObjects for importance parameter use
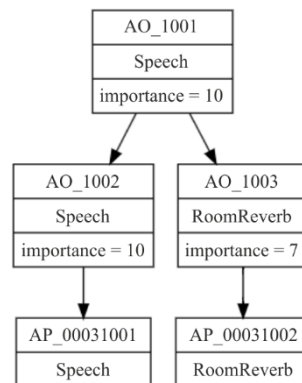
Report BS.2388-08

However, it is recommended that any audioObject with an importance of 10 be left untouched if possible.

3.12.4   Using an importance threshold

A processor that receives ADM files with the intention of reducing the number of tracks or objects can use a thresholding technique to achieve this reduction. The steps taken would be:

1) Set threshold to 0.

2) Inspect audioPackFormats and audioObject importance parameters. If any are less than the threshold then discard or ignore those elements.

3) Count the number of tracks and/or objects remaining.

4) If the number of tracks/objects is below the target then stop.

5) Else increase the threshold by 1 and repeat from step 2.

Other methods could be used for reducing ADM files, such as combining objects, intermediate rendering or downmixing.

4 Location of ADM metadata

The parent or top level element of the ADM is audioFormatExtended. Recommendation ITU-R BS.2076 does not specify the structure of other audio-relaed metadata with respect to audioFormatExtended. The audioFormatExtended element can be located within the ebuCoreMain element or within the other XML elements, as specified in Recommendations ITU-R BS.2088 and ITU-R BS.2125. Figure 10 illustrates such examples, and the following sections provide details pertaining to Recommendations ITU-R BS.2088 and ITU-R BS.2125.

FIGURE 10

Examples of location of the audioFormatExtended element
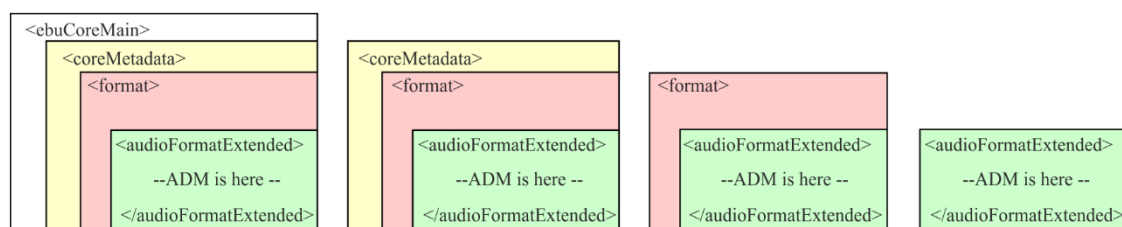
Report BS.2388-09

4.1 BW64 file specified in Recommendation ITU-R BS.2088

In BW64 files the ADM metadata is carried in the axml chunk, that may also contain other XML code. The ebuCoreMain element and the coreMetadata sub-element of the ebuCoreMain are used to describe information of the bext / ubxt chunk. In this case the audioFormatExtended element is located within the format element of the coreMetadata element as shown in Fig. 11.

FIGURE 11

**Location of the audioFormatExtended element specified in Recommendation ITU-R BS.2088**
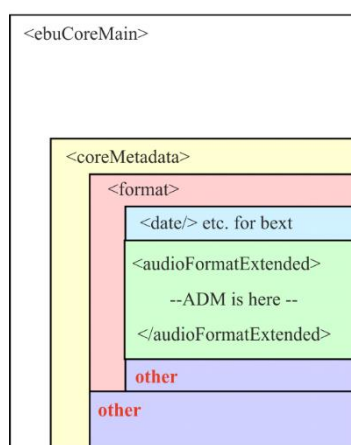
Report BS.2388-10

4.2 A serial representation of the ADM (S-ADM) specified in Recommendation ITU-R BS.2125

The frame element is the top level element of the S-ADM. When only audio-related metdata is used, the frame element directly includes the audioFormatExtended element, which includes the ADM metadata (excluding the frameHeader element). Alternatively, the frame element includes the coreMetadata element when broadcaster metadata and the other additional metadata are used.

Then, the format sub-element of the coreMetadata includes the audioFormatExtended element. Figure 12 shows the two structures of how the audioFormatExtended elment is carried in S-ADM.

FIGURE 12

Location of the audioFormatExtended element specified in Recommendation ITU-R BS.2125
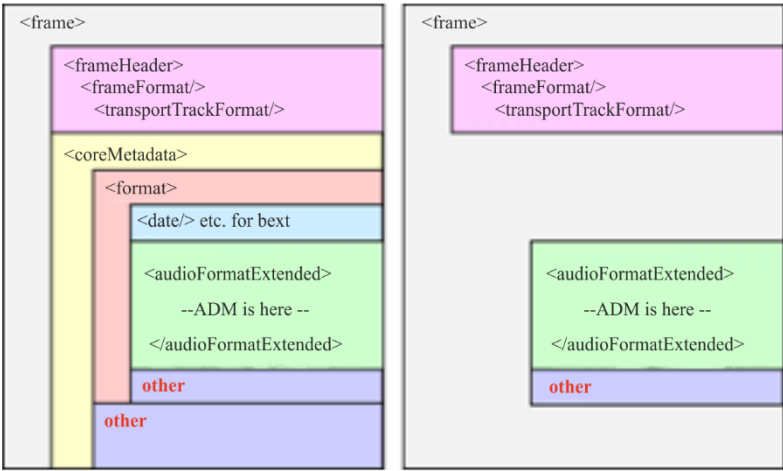
Report BS.2388-11

5   Worked examples

5.1   5.1 and Stereo combination

A common delivery configuration is to carry a 5.1 surround main mix alongside a stereo version to cater for non-5.1 compatible systems. In Recommendation ITU-R BS.1738, Production Scenario 5 describes such a configuration, as shown the table below.

Channel number	5.1 Surround sound audio signal
1	Left channel, complete mix
2	Right channel, complete mix
3	Centre channel, complete mix
4	Low frequency effects
5	Left surround channel
6	Right surround channel
7	Optional left channel international sound
8	Optional right channel international sound

This is straightforward to deal with by using the Common Definitions to describe the channels, and audioObjects to classify the two mixes. The <chna> can be generated like this:

Track number	audioTrackUID	audioChannelFormatID	audioPackFormatID
1	ATU_00000001	AC_00010001_00	AP_00010003
2	ATU_00000002	AC_00010002_00	AP_00010003
3	ATU_00000003	AC_00010003_00	AP_00010003
4	ATU_00000004	AC_00010004_00	AP_00010003
5	ATU_00000005	AC_00010005_00	AP_00010003
6	ATU_00000006	AC_00010006_00	AP_00010003
7	ATU_00000007	AC_00010001_00	AP_00010002
8	ATU_00000008	AC_00010002_00	AP_00010002

The tracks can now be connected to audioObjects so these need to be defined. The XML code for these two audioObjects is below:

```
<audioObject audioObjectID="AO_1001" audioObjectName="5.1_mix">
  <audioPackFormatIDRef>AP_00010003</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000001</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000002</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000003</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000004</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000005</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000006</audioTrackUIDRef>
</audioObject>

<audioObject audioObjectID="AO_1002" audioObjectName="stereo_mix">
  <audioPackFormatIDRef>AP_00010002</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000007</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000008</audioTrackUIDRef>
</audioObject>
```

More information can be added, as the two mixes may have some additional information. This information can be placed in the audioContent and audioProgramme elements:

```
<audioProgramme audioProgrammeID="APR_1001" audioProgrammeName="Complete+International">
  <audioContentIDRef>ACO_1001</audioContentIDRef>
  <audioContentIDRef>ACO_1002</audioContentIDRef>
</audioProgramme>

<audioContent audioContentID="ACO_1001" audioContentName="CompleteMix">
  <audioObjectIDRef>AO_1001</audioObjectIDRef>
</audioContent>

<audioContent audioContentID="ACO_1002" audioContentName="InternationalMix">
  <audioObjectIDRef>AO_1002</audioObjectIDRef>
</audioContent>
```

This is all the XML code that is required for the <axml> chunk: all the track, stream and channel information resides in the Common Definitions resource.

5.2 Object-based with a channel-based bed

This example shows how a pair of audio objects can be combined with a stereo channel-based bed. Figure 13 shows how the channels and objects are arranged (the timings are HH:MM).

Arrangement of channels and objects with corresponding timings
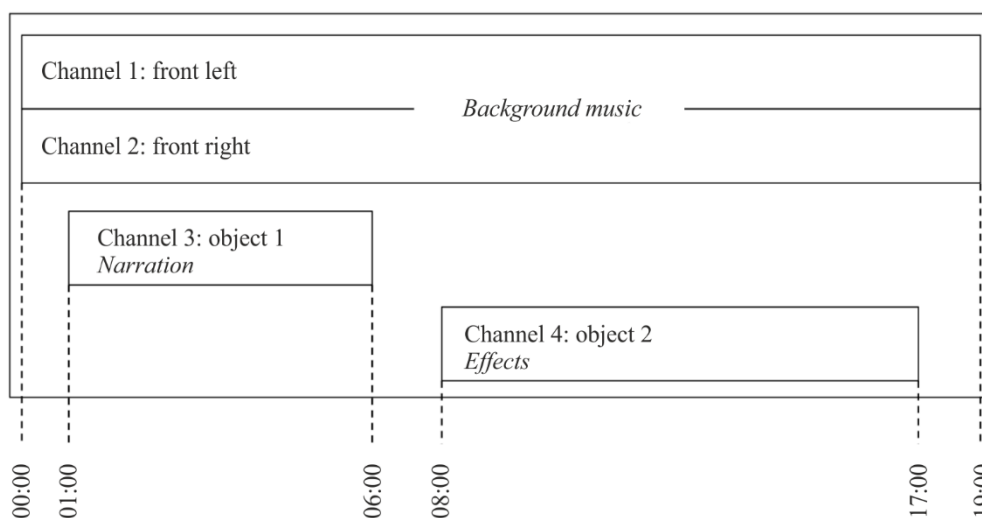
Report BS.2388-12

The background music part is a stereo channel-based bed that lasts the duration of the file. The first object is narration that lasts from 01:00 to 06:00. The second object is effects and lasts from 08:00 to 17:00, and this object is also dynamic as its position changes over time.

The channel-based channels can be described using common definitions, so those IDs need to be selected from the Common Definitions set of channels. The two objects will be custom definitions, so will need to be explicitly defined and included in the <axml> chunk.

The first stage is to define the channels for the two objects. The XML generated for the audioChannelFormat and audioBlockFormat elements is here:

```
<audioChannelFormat          audioChannelFormatID="AC_00031001"          audioChannelFormatName="Object1"
typeDefinition="Objects">
  <audioBlockFormat          audioBlockFormatID="AB_00031001_00000001"          rtime="00:00:00.00000"
duration="00:05:00.00000">
    <position coordinate="azimuth">0.0</position>
    <position coordinate="elevation">-10.0</position>
    <position coordinate="distance">1.0</position>
  </audioBlockFormat>
</audioChannelFormat>

<audioChannelFormat          audioChannelFormatID="AC_00031002"          audioChannelFormatName="Object2"
typeDefinition="Objects">
  <audioBlockFormat          audioBlockFormatID="AB_00031002_00000001"          rtime="00:00:00.00000"
duration="00:03:00.00000">
    <position coordinate="azimuth">-22.5</position>
    <position coordinate="elevation">5.0</position>
    <position coordinate="distance">1.0</position>
  </audioBlockFormat>
  <audioBlockFormat          audioBlockFormatID="AB_00031002_00000002"          rtime="00:03:00.00000"
duration="00:03:00.00000">
    <position coordinate="azimuth">0.0</position>
    <position coordinate="elevation">5.0</position>
    <position coordinate="distance">1.0</position>
  </audioBlockFormat>
  <audioBlockFormat          audioBlockFormatID="AB_00031002_00000003"          rtime="00:06:00.00000"
duration="00:03:00.00000">
    <position coordinate="azimuth">22.5</position>
    <position coordinate="elevation">5.0</position>
    <position coordinate="distance">1.0</position>
  </audioBlockFormat>
</audioChannelFormat>
```

The two objects also need audioPackFormat definitions. As both objects are single channels, these packs are just single channel ones:

```
<audioPackFormat audioPackFormatID="AP_00031001" audioPackFormatName="Object1" typeLabel="0003"
typeDefinition="Objects">
  <audioChannelFormatIDRef>AC_00031001</audioChannelFormatIDRef>
</audioPackFormat>

<audioPackFormat audioPackFormatID="AP_00031002" audioPackFormatName="Object2" typeLabel="0003"
typeDefinition="Objects">
  <audioChannelFormatIDRef>AC_00031002</audioChannelFormatIDRef>
</audioPackFormat>
```

The next stage is to prepare the <chna> chunk. As the two audio objects do not overlap in time it is possible for them to share the same track in the file. Here is an example <chna> chunk:

Track number	audioTrackUID	audioChannelFormatID	audioPackFormatID
1	ATU_00000001	AC_00010001_00	AP_00010002
2	ATU_00000002	AC_00010002_00	AP_00010002
3	ATU_00000003	AC_00031001_00	AP_00031001
3	ATU_00000004	AC_00031002_00	AP_00031002

The tracks and the packs can now be connected using audioObject elements. There are three audioObjects: one for the stereo channels, and one each for the two objects. The two objects will have start and duration times applied to them. The XML is shown here:

```
<audioObject audioObjectID="AO_1001" audioObjectName="StereoBed">
  <audioPackFormatIDRef>AP_00010002</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000001</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000002</audioTrackUIDRef>
</audioObject>

<audioObject audioObjectID="AO_1002" audioObjectName="ObjectNarration" start="00:01:00.00000"
duration="00:05:00.00000">
  <audioPackFormatIDRef>AP_00031001</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000003</audioTrackUIDRef>
</audioObject>

<audioObject audioObjectID="AO_1003" audioObjectName="ObjectEffects" start="00:008:00.00000"
duration="00:09:00.00000">
  <audioPackFormatIDRef>AP_00031002</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000003</audioTrackUIDRef>
</audioObject>
```

These audioObjects can be connected to audioContent elements. To add some extra information to the audioContents elements, the dialogue and loudness sub-elements have been added.

```
<audioContent audioContentID="ACO_1001" audioContentName="BackgroundMusic">
  <audioObjectIDRef>AO_1001</audioObjectIDRef>
  <dialogue dialogueContentKind="1">0</dialogue>
  <loudnessMetadata loudnessMethod="BS.1770">
    <integratedLoudness>-29</integratedLoudness>
  </loudnessMetadata>
</audioContent>

<audioContent audioContentID="ACO_1002" audioContentName="Narration">
  <audioObjectIDRef>AO_1002</audioObjectIDRef>
  <dialogue dialogueContentKind="2">1</dialogue>
  <loudnessMetadata loudnessMethod="BS.1770">
    <integratedLoudness>-21</integratedLoudness>
  </loudnessMetadata>
</audioContent>
```

```
<audioContent audioContentID="ACO_1003" audioContentName="Effects">
  <audioObjectIDRef>AO_1003</audioObjectIDRef>
  <dialogue dialogueContentKind="2">0</dialogue>
  <loudnessMetadata loudnessMethod="BS.1770">
    <integratedLoudness>-25</integratedLoudness>
  </loudnessMetadata>
</audioContent>
```

The three audioContent elements are connected together within the audioProgramme element at the top level:

```
<audioProgramme audioProgrammeID="APR_1001" audioProgrammeName="InterestingProgramme">
  <audioContentIDRef>ACO_1001</audioContentIDRef>
  <audioContentIDRef>ACO_1002</audioContentIDRef>
  <audioContentIDRef>ACO_1003</audioContentIDRef>
</audioProgramme>
```

The XML is now ready for the <axml> chunk, and so the audio file can be constructed with the <axml> and <chunks> and three tracks of PCM audio.

5.3   Sharing tracks from audioObjects to achieve different mixes

The ADM allows multiple audioObjects to reference the same audioTrackUID, and therefore the same audio track. This feature allows the different mixes to be made from the same audio tracks by using audioObject's gain parameter.

5.3.1   Basic example

To illustrate this approach the diagram in Fig. 14 shows how two mixes of a programme that share the same audioTrackUIDs (and therefore the same audio tracks) can be achieved. Each audioProgramme ("Default" and "Dia-boost") has its own audioContent elements for "Effects" and "Dialogue"; and each of these audioContent elements reference their own audioObject elements. For the "Default" audioProgramme the two audioObjects (AO_1001 and AO_1002) have their gains set to 1.0. For the "Dia-boost" audioProgramme the two audioObjects (AO_1001 and AO_1002) have their gains set to 0.5 and 2.0 respectively.

The audioObjects AO_1001 and AO_1011 (each representing the "Effects") both reference audioPackFormat AP_00031001 as they have the same format (the audioChannelFormat references have been omitted for clarity). They also both reference the same audioTrackUID ATU_00000001, which in turn references the same audio track carrying the "Effects" audio.

Similarly, the audioObjects AO_1002 and AO_1012 (each representing the "Dialogue") both reference audioPackFormat AP_00031002. They also both reference the same audioTrackUID ATU_00000002, which in turn references the same audio track carrying the "Dialogue" audio.

FIGURE 14

Two versions of a programme using the same tracks
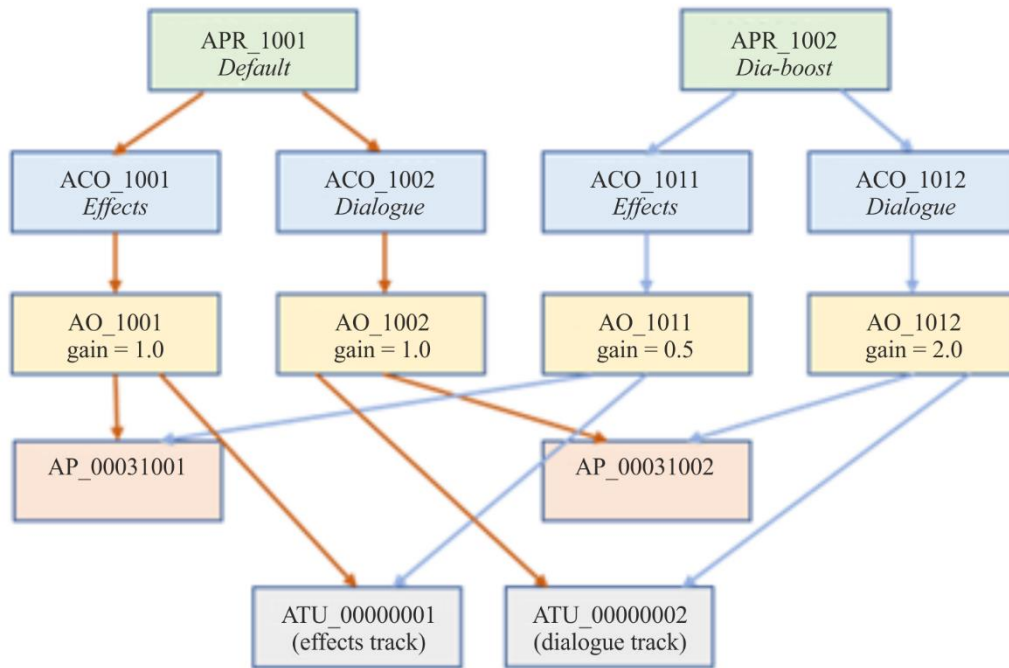
Report BS.2388-13

This approach allows audio tracks to be re-used in different mixes with different gains.

### 5.3.2 Using time parameters in audioObjects for different mixes

Another further feature of this approach is when dealing with different mixes where the audio in different tracks needs its gain adjusted differently over time depending on the levels in other objects. For example, if you have two different language versions of a programme with a common effects track, then the sound levels of the two dialogue tracks may differ. The difference in the dialogue tracks levels may require different mixing over time with the effects track. Therefore, the effects track may have to be referenced from two different audioObjects with different gains to handle these differences. Not only that, but the gains may have to vary over time, so the audioObjects will have to be divided into time-limited chunks.

Note, in this use-case, the gain transition between audioObjects is described as instantaneous. However, instantaneous changes of level may cause audible artifacts, therefore, gain interpolation or smoothing might need to be implemented (the behaviour is not specified here) in the renderer or playback device.

The diagram in Fig. 15 shows the basic layout of the ADM for this (audioPackFormat and below are omitted for clarity). Here the audioObjects AO_1001 and AO_1003 act as parents to time-limited audioObjects which can contain different gains. In this case AO_1032 has a gain of 0.6, whereas its corresponding audioObject, AO_1012, on the "English" side has a gain of 1.0.

FIGURE 15

Using time-limited audioObjects to adjust gain over time
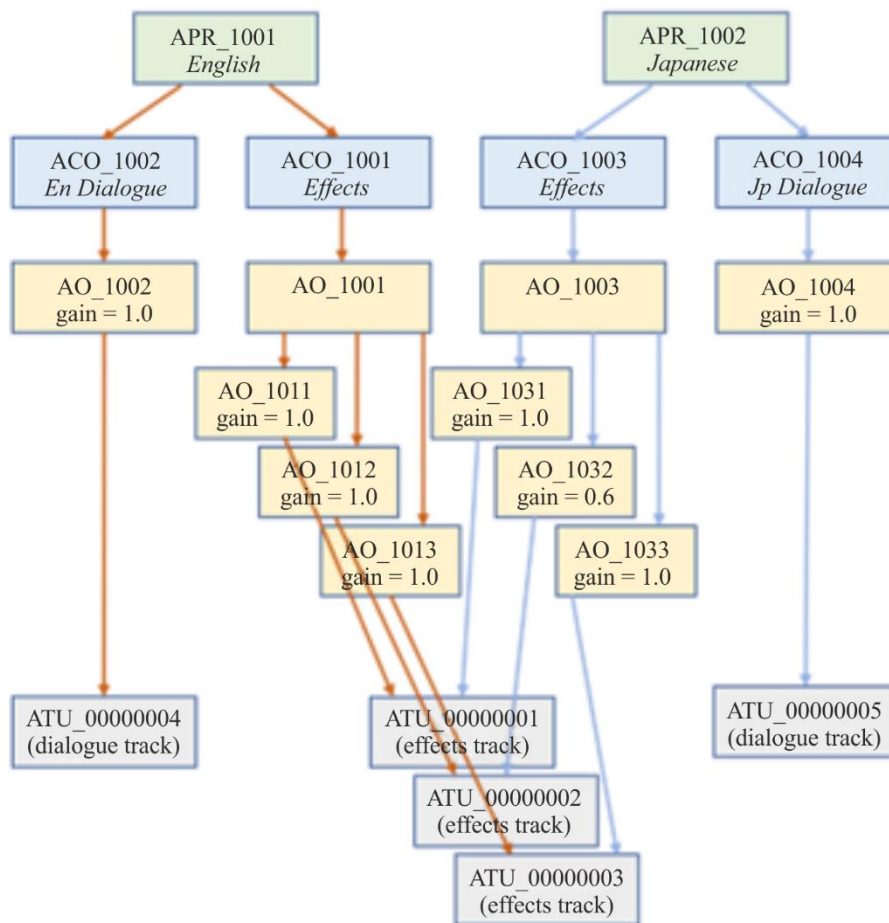
Report BS.2388-14

To get a better understanding of the time-related parameters are used, the diagram in Fig. 16 shows how the audioObjects are arranged on a time-line. AO_1002 and AO_1004, which carry the two dialogue objects, start at 0.0s and end at 100.0s. The green tracks show the envelopes of the signals in those tracks. In AO_1002 (ATU_00000004, track 1) the audio ducks at 50.0s. In AO_1004 (ATU_00000005, track 2) the audio ducks later at 70.0s. The effects audio in track 3 (ATU_00000001/2/3) is at low level from 0.0s to 50.0s (to duck under the dialogue in AO_1002) then rises from 50.0s to 100.0s. However, the raised level from 50.0s to 70.0s is too high for the dialogue in AO_1004, so this needs to be ducked. Therefore, AO_1032, which covers this 50.0s to 70.0s period provides a gain of 0.6 to duck this part of the effects track (as shown by the shading).

FIGURE 16

Time-based view of audioObjects and gain values
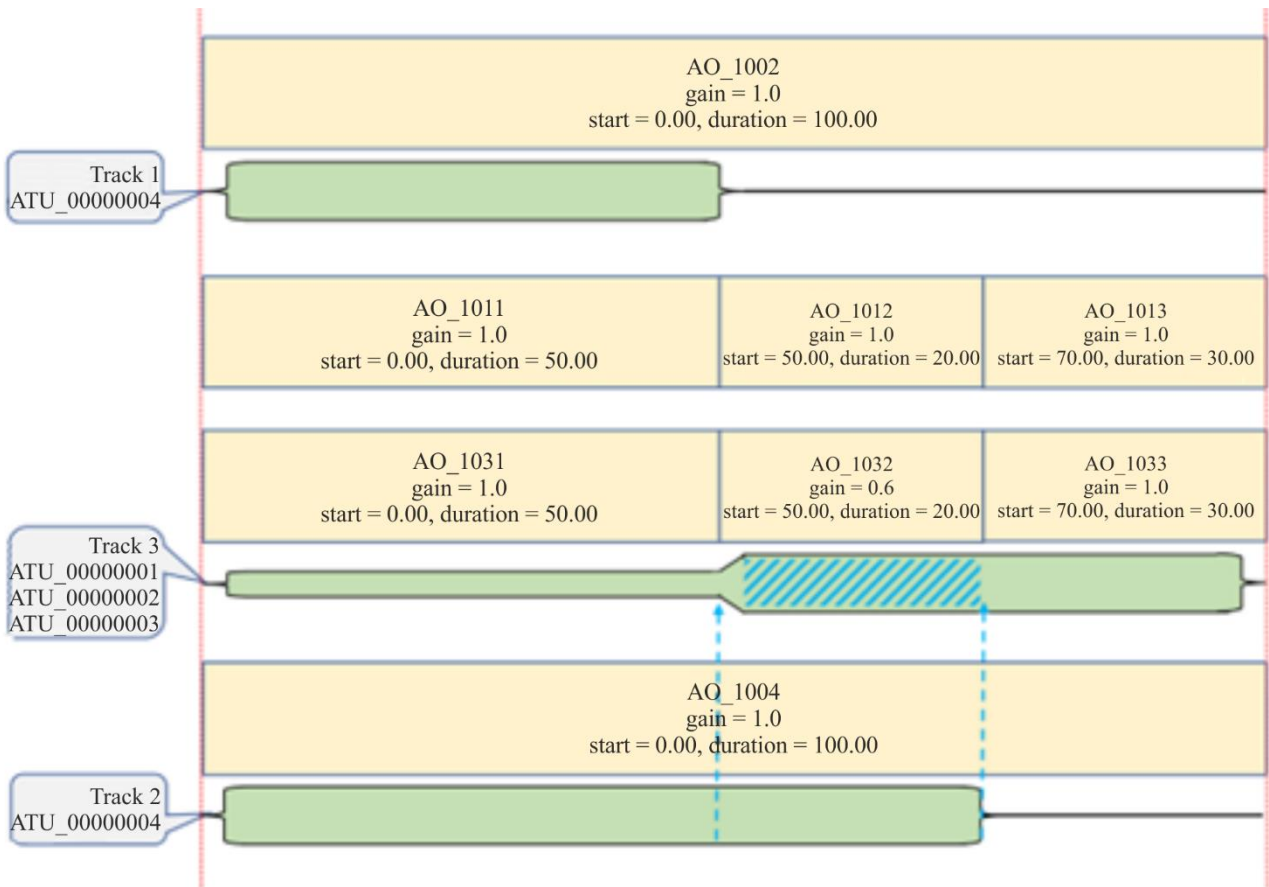
Report BS.2388-15

This approach does require knowledge of the levels of the audio in the tracks and splitting the audioObjects up accordingly.