# Book 6—Audio Support Libraries

## Part A:

# I/O and Control

TriMedia

Version 2.0 beta

# Table of Contents

# Chapter 7 SPDIF Renderer (ArendSpdif) API

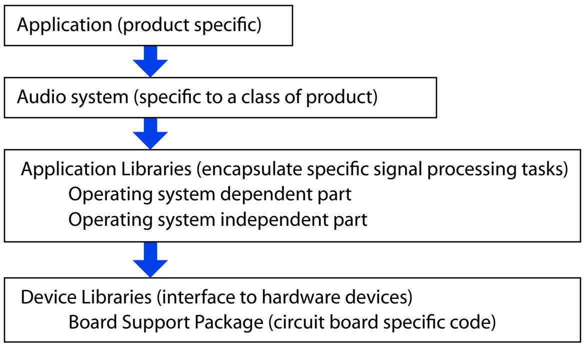
# Chapter 1

# TriMedia Audio Overview

# Introduction

The TriMedia application library set provides a rich set of building blocks for coding the audio portion of a multimedia application. The library defines an architecture and a framework, making it easier for you to create your own modules. This document provides an overview of how these blocks are integrated, and how you can use them to build your own audio systems.

The audio architecture is layered. Each layer encapsulates functionality up to a certain point. Because the interface to each layer is known and fixed, the implementation of any layer can be changed without affecting the other layers. A typical stack looks like this:


Application (product specific)
Audio system (specific to a class of product)
Application Libraries (encapsulate specific signal processing tasks)
Operating system dependent part
Operating system independent part
Device Libraries (interface to hardware devices)
Board Support Package (circuit board specific code)

The boundaries between these layers allow the most complex pieces of code to be reused in a flexible fashion. Specifically, the operating system independent portion of a signal-processing library might be used in many contexts. The context described here allows many such tasks to coexist in the media-processing environment.

## Device Libraries

The TriMedia chips include audio input and output devices as peripherals-on-chip. The device libraries provide a standard interface to these devices. A device library allows you to open, close, configure, start and stop the audio device(s). These are the names of the functions:

```
aiOpen              AoOpen
aiClose             AoClose
aiInstanceSetup     AoInstanceSetup
aiStart             AoStart
aiInstanceConfig    AoInstanceConfig
aiStop              aoStop
```

The device library does not specify a means of data transfer. Instead, its interface encourages you to install your own interrupt service routine to perform data transfer. When you examine the AI or AO hardware on TriMedia and compare it to the interface in the device library, you see that the device library provides a high level interface. The board support package (BSP) handles the lower level interface.

The audio device library is contained in a few files:

- tmaAI.c and tmAO.c contain the actual implementations.
- tmAI.h and tmAO.h are the public include files.
- tmAImmio.h and tmAOmmio.h are a collection of macros given in a standard form to simplify direct access to the hardware.
- tmAIboard.h and tmAOboard.h specify the interface between the device library and the BSP.

The device library is a very thin layer. It does little besides providing a platform-independent interface to the hardware. Because the use of the audio hardware is dependent upon the types of A/D and D/A converters present, the device library depends on a board support package (BSP) to describe access to the hardware.

## Board Support Package

Each of the device library functions has a companion function in the BSP. Library software accesses the BSP through a table of function pointers initialized at boot time. The table depends upon the hardware that is connected and supported. Refer to the software architecture book for more information.

Calling **aoInstanceSetup** specifies the operating mode for the audio device. Hence, the BSP's audio initialization function is called from **aoInstanceSetup** so the board can be configured for the requested operation. A typical example of this operation can be observed in the initialization function of the BSP for the TM-1300 debug board:

```
extern tmLibdevErr_t tm1300Debug_AO_Init( boardAOParam_t *param ){
   Float                 val;
   pprocCapabilities_t   procCap;
   Int                   err = TMLIBDEV_OK;
   Float                 tm1300DebugCPUClock;

/* get the clock frequency of the TriMedia CPU */
   err = procGetCapabilities(&procCap);
   if( err ) return err;
   tm1300DebugCPUClock = (Float) procCap->cpuClockFrequency;

/* reset the audio output peripheral */
   aoRESETM(AO_STATUS);

/* Set initial frequency to get AO into a stable state */
   val = param->sRate * 256.0;                      /* 256 * sampleRate */
   val = 0.5 + (477218588.0 * (val/tm1300DebugCPUClock));   /* 2**32 / 9 */
   aoSetFREQM( AO_STATUS, ((UInt)val)|0x80000000 );

   aoEnableSER_MASTERM(AO_STATUS);
   microsleep(10); /* wait until AO unit stabilized */
   if(!(param->audioTypeFormat & TM1300DEBUG_SUPPORTED_AUDIO_OUT_TYPES))
      return AIO_ERR_UNSUPPORTED_FORMAT;
   if(!(param->audioSubtypeFormat&TM1300DEBUG_SUPPORTED_AUDIO_OUT_SUBTYPES))
      return AIO_ERR_UNSUPPORTED_FORMAT;

#ifdef __LITTLE_ENDIAN__
   aoEnableLITTLE_ENDIANM(AO_STATUS);
#else
   aoDisableLITTLE_ENDIANM(AO_STATUS);
#endif
   aoMsbFirstM( AO_STATUS );
   aoStartRisingEdgeWSM( AO_STATUS );
   aoSampleRisingCLOCK_EDGEM( AO_STATUS ); /* sample on rising edge */
   aoSetSSPOSM( AO_STATUS, 0 );            /* L & R channel start at bit 0 */
   aoDisableSIGN_CONVERTM( AO_STATUS );    /* Disable Sign Convert for MSB */
   aoDisableWS_PULSEM( AO_STATUS );        /* word strobe in pulse mode */
   aoDisableTRANS_ENABLEM( AO_STATUS );

   switch (param->audioSubtypeFormat) {
      case apfStereo16:                 /* Two channel Audio */
         { /* 64 * 4 * 1 = 256fs */
         DP(("stereo 16 bit\n"));
         aoSetTRANS_MODEM ( AO_STATUS,  3 );
         aoSetWSDIVM      ( AO_STATUS, 63 );
         aoSetSCKDIVM     ( AO_STATUS,  3 );
         aoSetNR_CHANM    ( AO_STATUS,  0 );
         aoSetLEFTPOSM    ( AO_STATUS,  0 );
         aoSetRIGHTPOSM   ( AO_STATUS, 32 );
         aoSetSIZEM       ( AO_STATUS, param->size );
         break;
         }
      case apfStereo32:                 /* Two channel Audio, 32-bit */
         { /* 64 * 4 * 1 = 256 fs */
         DP(("stereo 32 bit\n"));
         aoSetTRANS_MODEM ( AO_STATUS,  1 );
         aoSetWSDIVM      ( AO_STATUS, 63 );
         aoSetSCKDIVM     ( AO_STATUS,  3 );
         aoSetNR_CHANM    ( AO_STATUS,  0 );
         aoSetLEFTPOSM    ( AO_STATUS,  0 );
         aoSetRIGHTPOSM   ( AO_STATUS, 32 );
         aoSetSIZEM       ( AO_STATUS, param->size );
         break;
         }

      /* -- other cases omitted here for clarity -- */

      case apfSevenDotOne16:           /* Eight Channel Audio */
         {
         DP(("eight channels 16 bit\n"));
         aoSetTRANS_MODEM ( AO_STATUS,  3 );
         aoSetWSDIVM      ( AO_STATUS, 63 );
         aoSetSCKDIVM     ( AO_STATUS,  3 );
         aoSetNR_CHANM    ( AO_STATUS,  3 );
         aoSetLEFTPOSM    ( AO_STATUS,  0 );
         aoSetRIGHTPOSM   ( AO_STATUS, 32 );
         aoSetSIZEM       ( AO_STATUS, param->size * 4 );
         break;
         }
      case apfSevenDotOne32:           /* Eight Channel Audio, 32-bit */
         {
         DP(("eight channels 32 bit\n"));
         aoSetTRANS_MODEM ( AO_STATUS,  1 );
         aoSetWSDIVM      ( AO_STATUS, 63 );
         aoSetSCKDIVM     ( AO_STATUS,  3 );
         aoSetNR_CHANM    ( AO_STATUS,  3 );
         aoSetLEFTPOSM    ( AO_STATUS,  0 );
         aoSetRIGHTPOSM   ( AO_STATUS, 32 );
         aoSetSIZEM       ( AO_STATUS, param->size * 4 );
         break;
         }
      default:   /* unsupported subtype */
         return(AIO_ERR_UNSUPPORTED_FORMAT);
   }
/* set sample rate */
   err = tm1300Debug_AO_SetSRate( param->sRate );
   if( err ) return err;
   return TMLIBDEV_OK;
} /* end of tm1300Debug_AO_Init() */
```

The required register settings are made in the BSP, and a set of MMIO macros simplify this process. For more information on audio BSPs, refer to Chapter 2.

## Application Libraries

Several audio libraries are available to help you construct audio systems on TriMedia. They share a common architecture, and fall into a number of categories:

| Category | Description |
|---|---|
| Digitizers | Data "sources" of audio streams captured from the outside world. |
| Renderers | Data "sinks" where digital audio data is translated to the outside world. |
| Mixers | A catch-all library for many types of audio signal processing. |
| Decoders | Accept a data stream in some compressed format and output PCM data |
| Encoders | Accept a data stream in PCM format and output in some compressed format. |
| Signal Processing | Code packaged for use in a mixer. |

### Architecture

The audio libraries all conform to the TriMedia Streaming Software Architecture (TSSA) as described, in great detail, in Book 3, *Software Architecture,* Part B. This means they can be configured to run as data-driven tasks in their own threads. Although a complete discussion of the TSSA architecture is complex, several simple points can be made here.

There are three important interfaces within the application libraries.

- The OL layer interface is the most common. Here, the operating system services (pSOS, by default) run the audio process in its own thread. The thread contains a loop that runs as long as data are available, unless the application gives a command to stop processing. The OL layer includes not only the data processing, but also the code to manage data buffering. The OL layer interface can take advantage of the large body of standard services to make the buffer management job less tedious. The OL layer defines the buffer transfer and memory allocation mechanisms in such a way that components can easily and reliably cooperate.
- The AL layer is designed to eliminate any dependency on an operating system. Whereas the key function of the OL layer is the start function, the key function at the AL layer is the **processData** function. The **processData** function does not incorporate buffer handling; it contains only the signal processing core of the code. Using the AL interface, you can easily construct a component that calls another component in its own thread. A system designer can use less buffer memory and thus achieve lower latency. The AL layer also reduces the overhead associated with tasks and task switching.
- Audio signal processing components can use another standardized API; the Audio Signal Processing (ASP) interface. ASP modules were designed for use in the framework of the type of audio mixers TriMedia-based television receivers use. You can think of the ASP interface as a special case of the AL layer **processData** function. While the typical **processData** function uses TSA data packets as its input and output, ASP functions use a buffer structure optimized for use inside a mixer. These subtle differences become clearer as you explore the structure of a mixer. The simple mixer (a demo) illustrates these concepts in use.

No matter which layer or API is used, TriMedia audio processing modules adhere to strict interface standards. One header file is exported to describe each public interface. Portions of the code that are not designed to be used publicly are hidden. Global variables and other forms of name-space pollution are assiduously avoided. These are basic concepts to object-oriented code.

## Filter Graphs

OL layer TSSA components can be interconnected, creating a filter graph. The arrangement of the filter graph is set up by code that connects the various modules. This might be application code, and it might be a library module referred to as an audio system.

## Modules Types

Whether selecting an existing module or creating a module, it helps to understand the existing classes of audio processors. Reusing these concepts can greatly speed development.

### Renderers

Renderer components consume data; as far as software is concerned, renderers have only input ports. A renderer accepts a data stream and presents a signal to the outside world through a D/A converter. To date, two audio renderers have been created. These use the TriMedia AO and SPDIF output ports, respectively, to present their data. Renderers can be considered TSSA device drivers. They are implemented using interrupt service routines.

The TriMedia audio renderers also have extensive synchronization functionality that can synchronize audio to video, or audio to any sort of external clock. Much of this functionality is implemented using the TSSA progress function which is called from the interrupt service routine. Regardless of the destination of the audio, the audio renderers share a similar interface allowing applications to easily redirect their output.

Because the renderers are built upon the device libraries, and because the device libraries depend on board support packages, it is easy to use the same renderer code on diverse types of hardware.

The audio renderers are best used through the OL interface. Although **ArendAO** can be used at the AL layer, the OL layer is always preferred.

Each of the audio renderers is designed to be operated as a clock master. The software expects to use the AO DDS (refer to the data book) to support the synchronization features.

### Digitizers

Digitizer components generate data; as far as software is concerned, digitizers have only output ports. A digitizer accepts a signal from the outside world and generates a TSSA data stream. The device libraries support the interface to the hardware. To date, the only audio digitizer available uses the TriMedia AI port. Like renderers, digitizers eventually rely on the board support package to supply the code used to initialize and control the hardware. The AI port can be connected to an A/D converter, or it can receive data from a digital audio input port such as an SPDIF receiver. When used with an SPDIF receiver, the AI port will run as a clock slave. The rest of the audio system will expect to lock to the AI clock using a software PLL. This facility is demonstrated in the sample program **exolCopyAudio**.

The **exolCopyAudio** example also demonstrates how the digitizer can receive an interrupt from the SPDIF receiver so it can respond to status changes from the digital source. Support for this facility also resides in the board support package.

## Mixers

In the TriMedia examples, the mixer module contains any form of post processing. The mixer has an OL layer TSSA interface, so it runs in its own task. The ASP component standards were developed to facilitate this sort of post processing. The AmixSimple library is provided with documentation and source code to demonstrate this architecture. The mixer architecture was developed to facilitate exchanging signal processing modules between projects. A mixer is an example of a component that is likely to be specific to a class of products. Although it might be possible to create a generic mixer, it has not been done. Mixer components—tone filters, sample rate converters, and dynamics processors—can be built for generic use. The ASP interface defines how to do this.

Digitizers and renderers always expect multi-channel audio streams to be interleaved. However, for the type of processing a mixer performs, it is often useful to work with non-interleaved streams. Therefore, the mixers each contain a de-interleave phase when data is received and an interleave phase when data is being prepared for transmission. This factoring makes sense only when there is enough processing between the de-interleave and the interleave. It is important to optimize the cache behavior of the code to achieve maximum efficiency.

## Decoders

Decoders receive data in a compressed format such as AC-3 or MPEG audio. They output normal PCM data. Because psychoacoustic compression algorithms are complex enough to take a few tens of MIPS, it is efficient and convenient to package them as OL layer components. The AC-3 and MPEG audio decoders share a number of features in their TSSA interfaces. They each use a progress function to indicate when a valid bitstream is acquired. They implement error functions to report on errors found in the bitstream. The AC-3 decoder has two outputs. The second output can produce a bitstream encoded to meet the IEC61937 standard for the transmission of compressed audio over an IEC60938 (SPDIF) connection. TriMedia audio decoders are designed to receive an MPEG packetized elementary stream (PES). In this case, the PES packets are time-stamped and the decoder transfers the time stamps to the decoded packets according to MPEG rules.

To date, effective decoders for Dolby Digital (AC-3) and for MPEG-1 layer 2 audio are readily available. Decoders for MPEG-1 layer 3 and for MPEG-2 AAC are under development. A decoder for g.723 is also available as part of the video conferencing stack.

## Encoders

Encoders receive PCM audio as an input and produce compressed audio data at their output. An example of an encoder is the MPEG-1 layer 2 encoder that is shipped on the applications disk. Dolby Digital and MP3 encoders are also available as prototypes.

### Signal Processing Libraries

Audio Signal Processing (ASP) libraries provide a convenient way to exchange less complex audio algorithms. They are a variant of the non-streaming AL layer TSSA interface. The data-driven, thread-based architecture defined by TSSA can be too “heavy” when the signal processing component in question only requires a few MIPS: the buffering used to connect TSSA components can require more memory and introduce more latency than is required. The AL layer’s process data function provides a way to avoid these issues. The ‘process data’ function gives a precedent for a functional interface to the module in question. Basic TSSA documentation assumes the ‘process data’ function will take TSA packets as its inputs and outputs. But supporting all the interleaved audio formats described by TSA leads to a significant portion of code that must be duplicated in each module. The ASP interface is a variant of the ‘process data’ interface, but does not use TSA packets. Instead, it uses a structure that allows audio channels to be organized as an array of pointers to de-interleaved channels.

ASP modules export the same **Open**, **Close**, **InstanceSetup**, **InstanceConfig** interface common to all TSA libraries. They show their similarity in the way the **processData** function is implemented.

Examples of ASP libraries in use today are the tone control of the DTV mixer, the loudness control of the DTV mixer, and the low-pass filter of the simple mixer example. Other components such as dynamic range compressors or 3D audio filters could easily be packaged as ASP modules.

## Audio Systems

In practice, some of these components must be interconnected to make a useful audio system. The code that connects these components is likely to be application-specific, but the application might represent a class of products. To take advantage of the similarity between audio systems in related products, the connected filter graph of the audio components can be packaged into an “audio system.” This is exactly what has been done for the digital television product class. The DTV audio system connects the Dolby Digital decoder to a mixer that supports ProLogic decode, tone, and loudness. The source can be selected to come from the MPEG demultiplexer, the analog input, or a digital (SPDIF) input. All the types of control that have been found necessary are incorporated into a common interface.

This concept is easily extended to other product spaces. The audio system you need might be unique. The TSSA architecture allows you to construct a system for reuse.

# Chapter 2

# Audio Board Support Packages

## Introduction

The TriMedia device libraries are supported by a board support package. As the software architecture documentation explains, the BSP allows system designers to change the audio interface hardware that is connected to TriMedia without adversely affecting the higher level audio I/O modules. The BSP interface defines the functionality expected of the hardware. The BSP functions are called by the audio device library.

## Writing a BSP for Audio

The best way to start writing a BSP for audio is to copy the BSP from one of the example boards that is shipped with the TriMedia SDE. The BSP for IREF is one such example. It is unusual in the way it supports the AD1847 codec. The BSP for the TM-1300 debug board shows how to support a simple A/D and D/A combination. That may be the most generally applicable example. The DTV reference board support packages demonstrate how to handle an SPDIF input.

When you bring up a board, a typical sequence of tests is as follows:

- iictest.out: Ensure that system recognizes your board and installs your BSP.
- sine.out: Ensure that the basic interface to the D/A works.
- sthru.out: Ensure that basic A/D interface works.
- exolArendAO: Verifies the operation of AO through the renderer.
- exolCopyAudio: Verifies operation of audio digitizer.

## Parameter Structures

The audio BSP interfaces are defined in two header files, namely tmAIboard.h and tmAOboard.h found in the TCS/include/tm1 directory. Several structures are defined in these files. This chapter document discusses them.

### boardAOParam_t

The **boardAOParam_t** structure is passed to the AO board initialization function.

```
typedef struct {
   tmAudioTypeFormat_t     audioTypeFormat;       /* audio type    */
   UInt32                  audioSubtypeFormat;    /* audio subtype */
   UInt32                  audioDescription;
   Float                   sRate;                 /* sample rate in Hz */
   Int                     size;                  /* #samples in buffers */
   tmAudioAnalogAdapter_t  output;                /* output select */
} boardAOParam_t, *pboardAOParam_t;
```

The members of this structure give the AO hardware all it needs to be configured. The format type and subtype are chosen from those listed in tmAvFormats.h. They are likely to be something like **atfLinearPcm** and **apfStereo16**, respectively, specifying stereo 16-bit PCM operation. To specify 8-channel 20-bit operation, the type and subtype would be **atfLinearPcm** and **apfSevenDotOne32**, with the description set to 20. When the data subtype is 32-bit, the description field specifies the precision of the data. Another data type that might be supported by common hardware is **atf1937**. This is appropriate if an SPDIF transmitter is connected to the IIS interface and is sending audio data encoded according to the IEC61937 specification.

The use of the sample rate field, given in Hertz, is self-evident. The size field is used to initialize the size register of the TriMedia processor's AO hardware. It is written directly into the size register by the functions that usse this structure.

The output field selects between multiple output devices that might be connected to the TriMedia processor. This field is not often used for output, but the analogous field in the input parameter structure is very useful, for instance, to select between a digital and an analog input.

### boardAIParam_t

The analogous structure used for input control is defined in tmAIboard.h:

```
typedef struct {
   tmAudioTypeFormat_t     audioTypeFormat;      /* data type */
   UInt32                  audioSubtypeFormat;   /* data subtype */
   UInt32                  audioDescription;
   Float                   sRate;                /* sample rate in Hz */
   Int                     size;                 /* #samples in buffers */
   tmAudioAnalogAdapter_t  input;                /* input */
} boardAIParam_t, *pboardAIParam_t;
```

This parameter structure is filled in by the **aiInstanceSetup** (or **aoInstanceSetup**) funciton. It is passed to the board's initialization function.

## BSP Config Structure

The board support package is used only by the corresponding device library. The device library accesses the BSP through a function table that must be filled in by the board designer. For audio, this function "table" is also defined in tmAI/Aoboard.h:

```
typedef struct {
   Char           codecName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t (*initFunc)(pboardAIParam_t param);
   tmLibdevErr_t (*termFunc)(void);

/* called from aiStart(). */
   tmLibdevErr_t (*startFunc)(void);

/* called from aiStop(). */
   tmLibdevErr_t (*stopFunc)(void);

/* called from aiSetSRate(). */
   tmLibdevErr_t (*setSRate)(Float sRate);

/* called from aiGetSRate(). */
/* Should return an accurate value from the hardware. */
   tmLibdevErr_t (*getSRate)(Float *sRate);
/* called from aiSetVolume() */
   tmLibdevErr_t (*setVolume)(Int lGain, Int rGain);
/* called from aiGetVolume() */
   tmLibdevErr_t (*getVolume)(Int * lGain, Int *rGain);
/* called from aiSetInput() */
   tmLibdevErr_t (*setInput)(tmAudioAnalogAdapter_t input);
/* called from aiGetInput() */
   tmLibdevErr_t (*getInput)(tmAudioAnalogAdapter_t *input);
/* a backdoor to support features not forseen in the initial design */
   tmLibdevErr_t (*configFunc)(UInt32 subAddr, Pointer value);
/* reports the format of the incomming audio. */
/* This is intended to be used with digital input (like S/PDIF) where */
/* the format of the incoming data is not known in advance. */
   tmLibdevErr_t  (*getFormat)(tmAudioFormat_t *format);
/* describes the properties of the audio in unit, */
/* this information will be reported by the aiGetCapabilities() function */
   UInt32         audioTypeFormats;
   UInt32         audioSubtypeFormats;
   UInt32         audioAdapters;
   intInterrupt_t intNumber;
   UInt32         mmioBase;
   Float          maxSRate;
   Float          minSRate;
   UInt32         gpioFirstPin;
   UInt32         gpioLastPin;
} boardAIConfig_t;
```

## initFunc

In the third line of the preceding structure, you can see the init function that is to be called from InstanceSetup. This function is to prepare all of the audio MMIO registers to communicate with the audio hardware on the board. The software expects these registers to be set using the macros defined in tmAImmio.h and tmAOmmio.h. The initialization function may also write to IIC or XIO locations to take the audio hardware out of reset or to initialize it in other ways. At the end of the initialization function, the audio input or output unit is stopped. Transmit or capture are not enabled.

Your application must place the address of this function in the table. The device library will assert if it the **initFunc** field is left Null.

## termFunc

The termination function (fourth line in the preceding structure) is called from the device library's close function. It shuts down the hardware completely.

Your application must place the address of this function in the table. The device library will assert if it the **termFunc** field is left Null.

## setSRate / getSRate

Different sorts of hardware use different means to set and get the sample rate. These function pointers allow a board designer to do what is appropriate. Many boards use the

DDSs that are available for AI and AO to implement these functions. However, both AI and AO might be driven by one DDS. Or perhaps both are driven by the VO DDS. Another possibility is that the audio codec supports a few selected sample rates and these are chosen by writing to some external register. All these situations can be accommodated by placing the appropriate code in these functions.

Your application must place the address of these function in the table. The device library will assert if it the **setSRate** or **getSRate** fields are left Null.

## setVolume / getVolume

Some codecs include their own volume controls. These functions allow board designers to accommodate volume control.

These functions are optional. If the board does not support volume control, you can leave the function pointers Null.

## getInput / setInput / getOutput / setOutput

These functions choose between multiple hardware paths when implemented on the board. More commonly used for the input, one of these functions can, for example, choose between a microphone and a line input, or an analog input and a digital input.

These functions are optional. You can leave the functions pointers Null.

## ConfigFunc

The configuration function allows you to implement features that are not defined in the standard set. Application-specified configuration commands must have their high bit set (OR with 0x80000000). The audio renderers, digitizers, and device libraries pass these commands directly to the hardware to be processed.

Your application must place the address of this function in the table. The device library will assert if it the **configFunc** field is left Null.

## getFormat

The **getFormat** function is called when the capabilities of the device are requested. It returns values to the **boardAIConfig_t** fields just below the getFormat function pointer. These include the audio types, subtypes, and adapters that are supported by the current hardware. Note that each of these data are bit fields that are designed to be OR'd together to specify a selection of supported formats.

Your application must place the address of this function in the table. The device library will assert if it the **getFormat** field is left Null.

### Others

The remaining variables set up the audio unit further. The interrupt number and the MMIO base differentiate between the two AI and AO units that are available on some TriMedia variants. The GPIO numbers are used in a similar fashion when the TriMedia variant supports general-purpose IO. The maximum and minimum sample rates are specified here.

## How Is This Used?

The board support package is designed to be called exclusively by the device library. It should not be called by other code. The functions declared in the BSP can be declared static so that they are not visible outside the file where they are defined. The functions are entered into the config table, and the config table is entered into the registry so that the device library can find it when it needs it.

# Chapter 3

# Audio Device Library

**Note**
For a general overview of TriMedia device libraries, see Chapter 5, *Device Libraries*, of Book 3, *Software Architecture*, Part A.

## Audio Device Library Overview

The Audio Device Library for TriMedia provides a low-0level interface to the audio hardware available on TriMedia. The Audio Device Library is designed to:

- Control the Audio-in and Audio-out hardware on the TriMedia processor.
- Control A/D and D/A converters attached to the TriMedia processor.
- Support the audio renderer, audio digitizer, and other audio systems on the TriMedia chip.
- Abstract the differences between the AI and AO units present in the various chips in the TriMedia family.

**Note**
See the appropriate TriMedia databook.

The Audio-in and Audio-out device libraries are nearly symmetrical. They provide a relatively simple interface: the device is opened, a few parameters are set, and then, when audio is started, the audio is serviced by interrupts. The address of the Interrupt Service Routine (ISR) is passed in with the Instance setup function. A few routines are also provided to control audio once it is running. These include setting the sample rate, and on devices which support it in hardware, setting the volume or selecting an input.

The two libraries can be used independently, provided that you are aware of any hardware limitations. For example, in the case of an analog input/output (I/O) device such as AD1847, both the audio input and output are performed by the same chip. Hence, the same sample rate is used by both the input and output.

The original TriMedia chips had only one AI and AO unit, and the interface unconsciously reflected this. Since some TriMedia variants now support multiple AI or AO units, the device library interfaces have been extended to support this. The new functions are postfixed with “M” to identify their applicability to multiple units. The previous single unit functions are now implemented in terms of these new functions, defaulting to the first unit for compatibility.

Most of the library functions return zero on success or nonzero error codes. You must check the error values returned by all of the initialization functions. Many functions check and report the use of sizes and alignments that the hardware cannot support.

The model implemented here does not mandate any specific data transfer mechanism. The APIs allow the user to install his own Interrupt Service Routine (ISR).

The TriMedia device libraries are designed to be used to create device drivers. Whereas device drivers are operating-system specific, the device libraries are generic. And whereas device drivers specify a data transfer mechanism, the device libraries leave the data transfer mechanism to the user.

The example applications show how the Audio device library can be used on its own without a traditional device-driver structure. In a given operating system, it may or may not be useful to create a standard device driver for this peripheral. However, if you decide to create a device driver, the Audio Device Library should be very helpful.

## Demonstration Programs

The Audio Device Library includes several demonstration programs, including sine, fplay, fplay6, and sthru (found in examples/peripherals/audio), avio (found in examples/ peripherals/avio), and patest (found in examples/peripherals/patest). The audio test programs demonstrate simple uses of the Audio Device Library. The sine program plays a sine wave through AO. The fplay program plays a sound file through AO. The fplay6 program is a variation of fplay where 6 channel mode is used. The sthru program uses AI and AO to capture sound from AI and play it out through AO (with option to capture to file).

## Using the Audio Device Library

This version of the API has been tested on the TM-1000 IREF and debug boards, as well as on DTV reference boards. On the IREF, it is recommended that you use the AD1847 as master of the IIS port with masterclock provided by the TM-1000 Direct Digital Synthesizer (DDS).

The TriMedia Audio Device Library is contained in the archived device library libdev.a. To use the Audio Device Library, you must include the tmAI.h and tmAO.h header files. The libdev.a device library is linked automatically.

**Note**
While developing programs using the device library, always use the debug version of the library (libdev_g.a). Numerous error conditions are trapped using asserts in the debug library.

The Audio Device Library depends upon the Board Support API, which is also included in libdev.a and is transparent to the user. If you want to change the hardware components on the board, see Chapter 2, *Audio Board Support Packages*.

## Limitations

You should be aware of the following hardware and/or software limitations:

- The modes currently implemented in the Philips IREF board support package are:
  - — Stereo 16-bit mode
  - — Six channel, “Five dot one,” 16-bit.
- The design of the IREF board makes it impossible for six channel output to work simultaneously with audio input. This could be changed with a different board design and an appropriate board support package. Similarly, the input and output

must be set to use the same mode. An attempt to request dissimilar modes for input and output causes an error to be returned by the Philips IREF board support package.

- The DTV reference boards support 8-channel operation in both 16- and 32-bit modes. The DTV boards do not support mono operation.
- Calculation of the sample rate is based on TriMedia's cycle clock.The software gets its definition of this clock from a global variable that is patched when the program is loaded. On Windows, this value is read from the Windows registry residing in the Windows directory. It can be set explicitly using the debugger. You must ensure that the value specified matches your hardware.
- When setting sample rates, consider the fact that the value for the DDS control register is computed in 32-bit math, possibly resulting in inaccuracies because of truncation. The "GetSampleRate" functions are designed to return the actual sample as set in hardware.

## Audio Input API Data Structures

This section describes the Audio Input API device library data structures. These data structures are defined in the tmAI.h header file.

| Name | Page |
|---|---|
| aiCapabilities_t | 33 |
| aiInstanceSetup_t | 35 |

## aiCapabilities_t

```
typedef struct {
   tmVersion_t       version;
   Int               numSupportedInstances;
   Int               numCurrentInstances;
   char              codecName[DEVICE_NAME_LENGTH];
   UInt32            audioTypeFormats;
   UInt32            audioSubtypeFormats;
   UInt32            audioAdapters;
   Float             minSRate;
   Float             maxSRate;
   intInterrupt_t    intNumber;
   UInt32            mmioBase;
} aiCapabilities_t, *paiCapabilities_t;
```

## Fields

| | |
|---|---|
| `version` | Version of the AO (AI) library module so that software can identify changes. |
| `numSupportedInstances` | Number of AO units in the hardware. |
| `numCurrentInstances` | Number of AO units currently in use. |
| `codecName[DEVICE_NAME_LENGTH]` | A string giving the name of the codec used for this unit. |
| `audioTypeFormats` | An OR'd value of all the audio type formats supported by this library. Audio type formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when **aiInstanceSetup** is called. |
| `audioSubtypeFormats` | An OR'd value of all the audio subtype formats supported by this library. Audio subtype formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when **aiInstanceSetup** is called. |
| `audioAdapters` | An OR'd value of all the audio adapters supported by this library. Audio subtype formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when **aiInstanceSetup** is called. |
| `minSRate` | Lowest supported sample rate. |
| `maxSRate` | Highest supported sample rate. |
| `intNumber` | AO interrupt. |
| `mmioBase` | AO MMIO base address. |

## Description

A pointer to this structure is returned by the **aiGetCapabilities** function. It will be updated with values from the board support package when **aiInstanceSetup** is called.

## aiInstanceSetup_t

```
typedef struct {
   void                   (*isr)(void);
   intPriority_t            interruptPriority;
   Bool                     overrunEnable;
   Bool                     hbeEnable;
   Bool                     buf1fullEnable;
   Bool                     buf2fullEnable;
   tmAudioTypeFormat_t      audioTypeFormat;
   tmAudioAnalogAdapter_t   input;
   UInt32                   audioSubtypeFormat;
   Float                    srate;
   Int                      size;
   Pointer                  base1;
   Pointer                  base2;
} aiInstanceSetup_t, *paiInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `isr` | Interrupt service routine (ISR). |
| `interruptPriority` | The interrupt priority for the AI ISR installed. |
| `overrunEnable` | Enable (or disable) the ISR overrun interrupt. |
| `hbeEnable` | Enable (or disable) the ISR highway bandwidth error interrupt. |
| `buf1fullEnable` | Enable (or disable) the ISR buffer1 full interrupt. |
| `buf2fullEnable` | Enable (or disable) the ISR buffer2 full interrupt. |
| `audioTypeFormats` | The audio type format selected. Audio type formats are defined in the file tmAvFormats.h. |
| `audioSubtypeFormats` | The audio subtype format selected.Audio subtype formats are defined in the file tmAvFormats.h. |
| `input` | Selects the input source (line in, mic in, digital in, etc.). The value **aaaNone** selects the default. |
| `srate` | Sample rate in Hz. |
| `size` | Size of buffers, in samples. |
| `base1` | Base address of buffer 1. |
| `base2` | Base address of buffer 2. |

### Description

This struct is used by the function **aiInstanceSetup** to read setup parameters. All fields are used on initial setup. Bases and size are only updated on the initial setup. ISR, priority and flags can be updated while running.

## Audio Output API Data Structures

This section presents the Audio Output device library data structures. These data structures are defined in the tmAO.h header file.

| Name | Page |
|---|---|
| aoCapabilities_t | 37 |
| aoInstanceSetup_t | 39 |

## aoCapabilities_t

```
typedef struct {
   tmVersion_t       version;
   Int               numSupportedInstances;
   Int               numCurrentInstances;
   char              codecName[DEVICE_NAME_LENGTH];
   UInt32            audioTypeFormats;
   UInt32            audioSubtypeFormats;
   UInt32            audioAdapters;
   Float             minSRate;
   Float             maxSRate;
   intInterrupt_t    intNumber;
   UInt32            mmioBase;
} aoCapabilities_t, *paoCapabilities_t;
```

## Fields

| | |
|---|---|
| `version` | Version of the AO (AI) library module. Used by software to identify changes. |
| `numSupportedInstances` | Number of units in the hardware. |
| `numCurrentInstances` | Number of units currently in use. |
| `codecName` | A string giving the name of the codec installed (which varies depending on the board used). |
| `audioTypeFormats` | An OR'd value of all the audio type formats supported by this library. Audio type formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when **aiInstanceSetup** is called. |
| `audioSubtypeFormats` | An OR'd value of all the audio subtype formats supported by this library. Audio subtype formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when **aiInstanceSetup** is called. |
| `audioAdapters` | An OR'd value of all the audio adapters supported by this library. Audio subtype formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when **aiInstanceSetup** is called. |
| `minSRate` | Lowest supported sample rate. |
| `maxSRate` | Highest supported sample rate. |
| `intNumber` | AO interrupt. |
| `mmioBase` | AO MMIO base address. |

## Description

A pointer to this structure is returned by the **aoGetCapabilities** function. It is updated with values from the board support package when aiInstanceSetup is called.

## aoInstanceSetup_t

```
typedef struct {
   void                     (*isr)(void);
   intPriority_t              interruptPriority;
   Bool                       underrunEnable;
   Bool                       hbeEnable;
   Bool                       buf1emptyEnable;
   Bool                       buf2emptyEnable;
   tmAudioTypeFormat_t        audioTypeFormat;
   UInt32                     audioSubtypeFormat;
   tmAudioAnalogAdapter_t     output;
   Float                      sRate;
   Int                        size;
   Pointer                    base1;
   Pointer                    base2;
} aoInstanceSetup_t, *paoInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `isr` | Interrupt service routine. |
| `interruptPriority` | The interrupt priority for the AO ISR installed. |
| `underrunEnable` | Enable interrupt sources. |
| `hbeEnable` | Enable HBE interrupt. |
| `buf1fullEnable` | Enable the ISR when buffer 1 is empty. |
| `buf2fullEnable` | Enable the ISR when buffer 2 is empty. |
| `audioTypeFormats` | The audio type format selected. Audio type formats are defined in the file tmAvFormats.h. |
| `audioSubtypeFormats` | The audio subtype format selected. Audio subtype formats are defined in the file tmAvFormats.h. |
| `output` | Selects the output, if multiple are available. The value **aaaNone** selects the default. |
| `srate` | Sample rate in Hz. |
| `size` | Size of buffers, in bytes, in samples. |
| `base1` | Base address of buffer 1. |
| `base2` | Base address of buffer 2. |

### Description

This struct is used by the function **aoInstanceSetup** to read setup parameters. All fields are used on initial setup. Bases and size are only updated on the initial setup. ISR, priority and flags can be updated while running.

## Audio Input API Functions

This section presents the Audio Input device library functions.

| Name | Page |
|---|---|
| aiGetCapabilities | 41 |
| aiGetCapabilitiesM | 42 |
| aiGetNumberOfUnits | 43 |
| aiOpen | 44 |
| aiOpenM | 45 |
| aiInstanceSetup | 46 |
| aiChangeBuffer1 | 47 |
| aiChangeBuffer2 | 47 |
| aiClose | 48 |
| aiStop | 49 |
| aiStart | 50 |
| aiSetInput | 51 |
| aiGetInput | 52 |
| aiSetVolume | 53 |
| aiGetVolume | 54 |
| aiSetSampleRate | 55 |
| aiGetSampleRate | 56 |
| aiGetFormat | 57 |
| aiConfig | 58 |

## aiGetCapabilities

```
tmLibdevErr_t aiGetCapabilities(
   paiCapabilities_t   *pCap
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a pointer to a structure of capabilities type. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assert is given if **pCAP** is NULL. |
| (other errors) | Error codes may be returned from call to board-GetConfig. See the boardGetConfig API documentation for possible error codes. |

### Description

This function fills in the value of a user-supplied pointer variable which will then point to the single shared capabilities structure for the AI device library. Implemented by a call to **aiGetCapabilitiesM** with **unitName** set to the first unit.

## aiGetCapabilitiesM

```
tmLibdevErr_t aiGetCapabilitiesM(
   paiCapabilities_t   *pCap,
   unitSelect_t         unitName
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a pointer to a structure of capabilities type. |
| `unitName` | Select which unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assert is given if **pCAP** is NULL. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | The selected unit is not supported in hardware. |

### Description

Used to find out about the AI hardware.

## aiGetNumberOfUnits

```
tmLibdevErr_t aiGetNumberOfUnits(
   UInt32  *pNumberOfUnits
);
```

### Parameters

| | |
|---|---|
| `pNumberOfUnits` | Pointer to an integer describing the number of audio input units that are supported on the current hardware. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assert is given if **pNumberOfUnits** is **NULL**. |

### Description

Used to find the number of audio input units that are supported on the current hardware.

## aiOpen

```
tmLibdevErr_t aiOpen(
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer to Int to hold instance value assigned on successful completion of the **aiOpen** function. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NO_MORE_INSTANCES` | There are no more free instances. |
| *(other errors)* | Error codes may be returned from the call to the intOpen function. See the Interrupt API documentation for possible error codes. |

### Description

This function will open an instance of the AI device and assign the instance value. It opens an interrupt (**intAUDIOIN**) with intOpen function. Implemented by a call to **aiOpenM** with **unitName** set to the first unit (**unit0**).

## aiOpenM

```
tmLibdevErr_t aiOpenM(
   Int           *instance,
   unitSelect_t   unitName
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer to Int to hold instance value assigned on successful completion of the **aiOpen** function. Used to access the unit in subsequent calls. |
| `unitName` | Select which unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Unit not supported in hardware. |
| `TMLIBDEV_ERR_NO_MORE_INSTANCES` | No more free instances. Unit not supported in hardware. |
| `TMLIBDEV_ERR_MEMALLOC_FAILED` | Memory used for instance variable structure. |

Other errors might be returned if the allocation of the interrupt or hardware pins (on GPIO enabled devices) fail.

### Description

This function opens an instance of the AI device and assigns the instance value. It opens an interrupt (**intAUDIOIN**) with **intOpen** function. Implemented by a call to **aiOpenM** with **unitName** set to the first unit (**unit0**).

## aiInstanceSetup

```
tmLibdevErr_t aiInstanceSetup(
   Int                    instance,
   paiInstanceSetup_t    pSetup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of **the aiOpen** function. |
| `pSetup` | Pointer to setup structure containing setup parameters. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered instead of an error code. |
| `AIO_ERR_UNSUPPORTED_FORMAT` | The requested audio format is not supported by the AI library. |
| `BOARD_ERR_NULL_FUNCTION` | The board codec struct for AI is missing an initialization function or the setSRate function. |

Other error codes may be returned due to call to **boardGetConfig**. See the board API for details. Other error codes may also be returned because of the call to **intInstanceSetup**. See the interrupts API for details. Other error codes may also be returned due to call to the board codec initialization function or the setSRate function. This is specific to the board codec.

### Description

The aiInstanceSetup function performs initialization of the AI hardware. This function will get the setup function from the board API by **boardGetConfig**. Then it runs the appropriate setup function (see board API), setups the AI_CTL MMIO register according to **pSetup**. It sets the base and size MMIO registers with **aiSetBASE1**, **aiSetBASE2**, and **aiSetSIZE** macros, then it setups the interrupt opened by aiOpen with intInstanceSetup. Upon return, the device is stopped with **aiDisableCAP_ENABLE** macro, but the device is ready to go (via **aiStart**).

## aiChangeBuffer1

```
void aiChangeBuffer1(
   Int       instance,
   Pointer   buffer
)
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of **the aiOpen** function. |
| `buffer` | New buffer pointer. |

### Return Codes

There are no return codes because this function is implemented as a macro.

### Description

This function (macro) changes the base address of buffer1.

## aiChangeBuffer2

```
void aiChangeBuffer2(
   Int       instance,
   Pointer   buffer
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of **the aiOpen** function. |
| `buffer` | New buffer pointer. |

### Return Codes

There are no return codes because this function is implemented as a macro.

### Description

This function (macro) changes the base address of buffer2.

## aiClose

```
tmLibdevErr_t aiClose(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `TMLIBDEV_ERR_NOT_OWNER` | In the debug version of the library, this assertion is triggered if the instance does not match the owner. |
| *(other error codes)* | Other error codes might be returned due to the call to **intClose**. Please see the Interrupt library API for the possible returned error codes. |

### Description

Close the given instance of the AI device, after which the device is free and ready for allocation. This function shuts down driver by calling the appropriate termination function from the board API, and interrupt service through MMIO AI_CTL register. Then it closes with **intClose** the interrupt **intINTAUDIOIN** opened by **aiOpen**.

## aiStop

```
tmLibdevErr_t aiStop(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match the owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |

### Description

This function stops the audio-in capture for the given instance, by calling the **aiDisableCAP_ENABLE** macro (see tmAImmio.h).

## aiStart

```
tmLibdevErr_t aiStart(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | If **aiSetInput** or **aiSetVolume** was called while audio is stopped, then **aiStart** will make calls to the board codec **setInput** and **setVolume** functions. If those functions are missing in the board codec struct, this error is returned. |
| *(other error codes)* | If **aiSetInput** or **aiSetVolume** was called while audio is stopped, then aiStart will make calls to the board codec **setInput** and **setVolume** function. Other error codes may be returned by those board codec functions. |

### Description

This function starts the audio in capture for the given instance. It calls the macro **aiEnableCAP_ENABLE**, and will adjust the input and the volume when needed with the appropriate board functions (see board API).

## aiSetInput

```
tmLibdevErr_t aiSetInput(
   Int                     instance,
   tmAudioAnalogAdapter_t  input
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |
| `input` | Codec input selection. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | The board codec struct does not contain a **setInput** function. |
| (other error codes) | Other error codes may be returned by the board codec setInput function, which is called by **aiSetInput**. |

### Description

This function is used to set or change codec input selection. This function calls the appropriate board function (see board API). If the device is stopped, this call is postponed until the next **aiStart**.

## aiGetInput

```
tmLibdevErr_t aiGetInput(
   Int                        instance,
   tmAudioAnalogAdapter_t   *input
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |
| `input` | Pointer to caller's struct to be filled. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assert is triggered if the input is NULL. |

### Description

This function is used to get the current codec input selection.

## aiSetVolume

```
tmLibdevErr_t aiSetVolume(
   Int   instance,
   Int   lgain,
   Int   rgain
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |
| `lgain` | Left channel gain, expressed in 1/100th of a decibel (dB). |
| `rgain` | Right channel gain, expressed in 1/100th of a decibel (dB). |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | The board codec struct is missing a **setVolume** function. |

Other error codes may be returned because of the call to the board codec **setVolume** function. This is specific to the board codec function used.

### Description

This function is used to set or change the audio input gain. This is achieved by calling the appropriate setVolume function when this function is installed (look at board API). If the Audio In is stopped, this is postponed until the next **aiStart** call.

### Implementation Notes

For the ad1847 codec on the TriMedia IREF board, valid input volume ranges from 0.0 dB to +22.5 dB inclusive, in increments of +1.5 dB. This corresponds, for example, to an lgain value range of 0 to 2250 in increments of 150. Values within the range will be adjusted down to the lower closest legal value, and values outside of range will result in non-zero error codes being returned.

## aiGetVolume

```
tmLibdevErr_t aiGetVolume(
   Int    instance,
   Int   *lgain,
   Int   *rgain
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |
| `lgain` | Pointer to variable in which to return the left channel gain. |
| `rgain` | Pointer to variable in which to return the right channel gain. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered when lgain and/or rgain is Null. |

### Description

This function is used to get the audio input gain.

## aiSetSampleRate

```
tmLibdevErr_t aiSetSampleRate(
   Int     instance,
   Float   srate
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |
| `srate` | Sample rate. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `AIO_ERR_SRATE_TOO_HIGH` | The requested sample rate is higher than the maximum allowed by the chosen codec. |
| `AIO_ERR_SRATE_TOO_LOW` | The requested sample rate is lower than the minimum allowed by the chosen codec. |
| `BOARD_ERR_NULL_FUNCTION` | The board codec struct is missing the setSRate function. |
| *(other error codes)* | Other error codes may be returned by the board codec setSRate function, which is board codec specific. |

### Description

This function sets or changes the sample rate, by calling the appropriate function **SetSRate** from the board API when this function is installed (see board API).

## aiGetSampleRate

```
tmLibdevErr_t aiGetSampleRate(
   Int     instance,
   Float  *srate
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |
| `srate` | Pointer to variable in which to return the sample rate. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered if srate is NULL. |
| `BOARD_ERR_NULL_FUNCTION` | The board codec structure is missing the getSRate function. |
| *(other error codes)* | Other error codes may be returned by the board codec function **getSRate**. This is specific to the board codec function. |

### Description

This function gets the current sample rate.

## aiGetFormat

```
tmLibdevErr_t aiGetFormat(
   Int                instance,
   tmAudioFormat_t   *format
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |
| `format` | Pointer to a format structure in which to return information about the current format. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | The codec does not support Configure. |
| *(other error codes)* | Other error codes may be returned by the board-specific codec Configure function. |

### Description

This routine is provided to support the implementation of digital audio input. It will return the format of the currently selected audio input. When the digital input has been selected (using **aiSetInput**), this function (via the BSP) will query the digital audio input receiver and report the detected format. In the case of an SPDIF receiver, the sample rate, the bit depth, and the “non-PCM” data bit may be used to indicate whether this is PCM audio data or encoded AC-3.

### Implementation Notes

Implementation is board specific.

## aiConfig

```
tmLibdevErr_t aiConfig(
   Int      instance,
   UInt32   subaddr,
   UInt32   value
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aiOpen** function. |
| `subaddr` | Generic pointer passed to codec Configure routine. |
| `value` | Generic pointer passed to codec Configure routine. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | The codec does not support Configure. |
| *(other error codes)* | Other error codes may be returned by the board-specific codec Configure function. |

### Description

This routine is provided to allow for passing of configuration information from the user program through the device library to the (customized) user board codec routines (look at board API). The two parameters are passed down to the board codec Configure routine without any processing. The error code returned will be the return value of the board codec Configure routine.

### Implementation Notes

For the Philips provided boards, the board codecs do not provide a Configure function, and therefore aiConfig will return an error. The config function is used in the DTV boards: It is used to talk to the SPDIF decoder.

## Audio Output API Functions

This section presents the Audio Output device library functions.

## aoGetCapabilities

```
tmLibdevErr_t aoGetCapabilities(
   paoCapabilities_t   *pCap
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return a pointer to capabilities data. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered if **pCap** is NULL. |
| *(other errors)* | Error codes may be returned from call to board-GetConfig. See the boardGetConfig API documentation for the possible error codes. |

### Description

This function fills in the value of a user-supplied pointer variable which will then point to the single shared capabilities structure for the AO device library. Implemented by a call to **aoGetCapabilitiesM** with **unitName** set to the first unit (**unit0**).

## aoGetCapabilitiesM

```
tmLibdevErr_t aoGetCapabilitiesM(
   paoCapabilities_t   *pCap,
   unitSelect_t         unitName
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a pointer to an AO capabilities structure. |
| `unitName` | Select which unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Unit not supported in hardware. |

### Description

Finds out about the AO hardware.

## aoGetNumberOfUnits

```
tmLibdevErr_t aoGetNumberOfUnits(
   UInt32  *pNumberOfUnits
);
```

### Parameters

| | |
|---|---|
| `pNumberOfUnits` | Pointer to a variable in which to return the number of audio output units that are supported on the current hardware. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assert is given if **pNumberOfUnits** is **NULL**. |

### Description

Finds the number of audio output units that are supported on the current hardware.

## aoOpen

```
tmLibdevErr_t aoOpen(
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer to Int to hold instance value assigned on successful completion of the **aoOpen** function. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NO_MORE_INSTANCES` | There are no more free instances. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered if the instance is NULL. |
| *(other errors)* | Error codes may be returned from call to **intOpen**. see Interrupt API documentation for possible error codes. |

### Description

This function opens an instance of the AO device. It opens the Audio Out interrupt (intAUDIOOUT) with the intOpen function. Implemented by a call to **aoOpenM** with **unitName** set to the first unit (**unit0**).

## aoOpenM

```
tmLibdevErr_t aoOpenM(
   Int           *instance,
   unitSelect_t   unitName
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer to instance variable, stored as an integer. This variable, assigned in open, is used to access the unit in subsequent calls. |
| `unitName` | Select which unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Not supported in hardware. |
| `TMLIBDEV_ERR_NO_MORE_INSTANCES` | Unit is already in use. |
| `TMLIBDEV_ERR_MEMALLOC_FAILED` | Memory used for instance variable structure. |

Other errors might be returned if the allocation of the interrupt or hardware pins (on GPIO enabled devices) fail.

### Description

This function will open an instance of the selected AO device and assign the instance value. It opens an interrupt as appropriate for the specified unit with **intOpen** function.

## aoInstanceSetup

```
tmLibdevErr_t aoInstanceSetup(
   Int                 instance,
   paoInstanceSetup_t  pSetup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |
| `pSetup` | Pointer to setup structure. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered instead of an error code. |
| `AIO_ERR_UNSUPPORTED_FORMAT` | The requested audio format is not supported by the AO library. |
| `BOARD_NULL_FUNCTION` | The board codec structure is missing the initialization function or the board codec structure is missing the setSRate function. |
| *(other error codes)* | Other error codes may be returned by the call to boardGetConfig. See the board API documentation for details.<br>Other error codes may be returned by the call to intInstanceSetup. See the interrupts API documentation for details.<br>Other error codes may be returned by the call to the board codec initialization function and the setSRate function. This is specific to the board codec. |

### Description

This function performs initialization of the AO hardware by calling the appropriate init_func from the board API returned by the boardGetConfig function. It setups the MMIO registers AO_CTL, AO_BASE1 and AO_BASE2 (**aoSetBASE1** and **aoSetBASE2** macros), and AO_SIZE (**aoSetSize** macro). Then it sets up, when needed, the Interrupt Service Routine by calling **intInstanceSetup**.

Upon return, the device is stopped but ready to go (via **aoStart**).

## aoChangeBuffer1

```
void aoChangeBuffer1(
   Int       instance,
   Pointer   buffer
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of **the aoOpen** function. |
| `buffer` | New buffer pointer. |

### Return Codes

This function has no return code because it is implemented as a macro.

### Description

This function changes the base address of buffer1.

## aoChangeBuffer2

```
void aoChangeBuffer2(
   Int       instance,
   Pointer   buffer
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of **the aoOpen** function. |
| `buffer` | New buffer pointer. |

### Return Codes

This function has no return code because it is implemented as a macro.

### Description

This function changes the base address of buffer2.

## aoClose

```
tmLibdevErr_t aoClose(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner.In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| *(other error codes)* | Other error codes may be returned by the call to intClose. See the Interrupts API document for possible return codes.<br>Other error codes may also be returned by the call to the board codec termination function (term_func). Those return values are specific to the board codec. |

### Description

This function will close the instance of the AO device by calling the appropriate termination function from the board API, and close with intClose the interrupt opened by **aoOpen**. After this function, the device is free and ready for re-allocation.

## aoStop

```
tmLibdevErr_t aoStop(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. |

### Description

This function stops the audio play of this instance. This is achieved by calling the **aoDisableTRANS_ENABLE** macro.

## aoStart

```
tmLibdevErr_t aoStart(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner.In the debug version of the library, this assertion is triggered if the instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | If **aoSetVolume** and/or **aoSetOutput** is called while audio is stopped, the board codec routines are called at **aoStart** time. The error is returned if the board codec structure is missing the **set-Output** or **setVolume** functions. |
| (other error codes) | If **aoSetVolume** and/or **aoSetOutput** is called while audio is stopped, the board codec routines are called at **aoStart** time. other error codes may be returned by the board codec functions **set-Output** and/or **setVolume**. The return codes in that case are board codec specific. |

### Description

The **aoStart** function starts the audio play of this instance by calling the **aoEnable-TRANS_ENABLE** macro. This function will set the volume and the output when needed (see **aoSetVolume** and **aoSetoutput**).

## aoSetOutput

```
tmLibdevErr_t aoSetOutput(
   Int                      instance,
   tmAudioAnalogAdapter_t   output
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |
| `output` | Codec input selection. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if the instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | The board codec structure is missing the set-Output function. |
| *(other error codes)* | Other error codes may be returned by the board codec setOutput function. These are specific to the board codec. |

### Description

This function is used to set or change codec output selection. It calls the appropriate function setOutput from the board API. This call is postponed until **aoStart** is called if Audio Out is stopped.

## aoGetOutput

```
tmLibdevErr_t aoGetOutput(
   Int                       instance,
   tmAudioAnalogAdapter_t   *output
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |
| `output` | Pointer to struct to be filled. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if the instance does not match the owner. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered if the output is NULL. |

### Description

This function is used to get the current codec output selection.

## aoSetVolume

```
tmLibdevErr_t aoSetVolume(
   Int    instance,
   Int    lgain,
   Int    rgain
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |
| `lgain` | Left channel gain, expressed in 1/100 of a decibel (dB). |
| `rgain` | Right channel gain, expressed in 1/100 of a decibel (dB). |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | Returned if instance does not match owner. In the debug version of the library, this assertion is triggered if the instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | Returned if the board codec structure is missing the setVolume function. |
| (other error codes) | Other error codes may be returned due to the call to the board codec setVolume function, and are specific to the board codec. |

### Description

This function is used to set or change the audio output gain. This calls the appropriate function setVolume from the board API. This call is postponed until **aoStart** is called if the Audio Out is stopped.

### Implementation Notes

For the ad1847 codec on the TriMedia IREF board, the valid input volume ranges from 0.0 dB to +22.5 dB inclusive, in increments of +1.5 dB. This corresponds to (for example) lgain value range of 0 to 2250 in increments of 150. Values within the range will be rounded down to the lower closest legal value, and values outside of range will result in non-zero error codes being returned.

## aoGetVolume

```
tmLibdevErr_t aoGetVolume(
   Int    instance,
   Int   *lgain,
   Int   *rgain
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |
| `lgain` | Pointer to a variable in which to return the left channel gain. |
| `rgain` | Pointer to a variable in which to return the right channel gain. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner.In the debug version of the library, this assertion is triggered if the instance does not match the owner. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered if lgain and/or rgain is Null. |

### Description

This function gets the audio output gain.

## aoSetSampleRate

```
tmLibdevErr_t aoSetSampleRate(
   Int     instance,
   Float   srate
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |
| `srate` | Sample rate. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match the owner. In the debug version of the library, this assertion is triggered if the instance does not match the owner. |
| `AIO_ERR_SRATE_TOO_HIGH` | The requested sample rate is higher than the maximum allowed by the board codec chosen. |
| `AIO_ERR_SRATE_TOO_LOW` | The requested sample rate is lower than the minimum allowed by the board codec chosen. |
| `BOARD_ERR_NULL_FUNCTION` | The board codec structure is missing the setSRate function. |
| *(other error codes)* | Other error codes may be returned due to the call to the board codec setSRate function, and are specific to the board codec. |

### Description

This function sets or changes the sample rate by calling the appropriate function setRate from the board API.

## aoGetSampleRate

```
tmLibdevErr_t aoGetSampleRate(
   Int     instance,
   Float   srate
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |
| `srate` | Pointer to Float to hold the data. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match the owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered if srate is Null. |
| `BOARD_ERR_NULL_FUNCTION` | The board codec structure is missing the getSRate function. |
| *(other error codes)* | Other error codes may be returned by the call to the board codec getSRate function, and are specific to the board codec. |

### Description

This function gets the current sample rate by calling the appropriate function getSRate from the board API.

## aoConfig

```
tmLibdevErr_t aoConfig(
   Int      instance,
   UInt32   subaddr,
   UInt32   value
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value assigned at the call of the **aoOpen** function. |
| `subaddr` | Generic pointer passed to the codec Configure routine. |
| `value` | Generic pointer passed to the codec Configure routine. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | The instance does not match owner. In the debug version of the library, this assertion is triggered if instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | The codec does not support Configure. |
| *(other error codes)* | Other error codes may be returned by the board codec Configure function, and are specific to the board codec. |

### Description

This routine is provided to allow for the passing of configuration information from the user program through the device library to the (customized) user board codec routines (look at the board API). The two parameters are passed down to the board codec Configure routine without any processing. The error code returned will be the return value of the board codec Configure routine.

### Implementation Notes

For the Philips provided boards, the board codecs do not provide a Configure function, and therefore **aoConfig** will return an error. The DTV reference boards use the config function to communicate with the SPDIF receiver.

# Chapter 4

# SPDIF Output Device Library

**Note**
For a general overview of TriMedia device libraries, see Chapter 5, *Device Libraries,* of Book 3, *Software Architecture*, Part A.

## SPDO API Overview

Some variants of the TriMedia chip (TM-1300, TM-2700) include an SPDIF output unit, and this hardware block is termed SPDO. S/P DIF is short for Sony/Philips Digital Interface Format. It is a digital audio exchange method. SPDIF is described by the international standard IEC60958 (formerly IEC958), which is a superset of the description provided by AES-3, the standard of the Audio Engineering Society.

The SPDO API described here is a low-level, device library API that controls the SPDO hardware. As such, it implements the usual Open, InstanceSetup, Start type of interface, but this level of the interface does not specify the data transfer method.

**Note**
The SPDO hardware is described in the hardware data books for the TM-2700 and the TM-1300.

The SPDO device library provides a relatively simple interface: the device is opened, a few parameters are set, and then, when audio is started, the audio is serviced by interrupts. The address of the Interrupt Service Routine (ISR) is passed in with the InstanceSetup function. A few functions are also provided to control audio once it is running. These include functions that set the sample rate.

Most of the library functions return either zero, on success, or nonzero error codes. You must check the error values returned by all of the initialization functions. Many functions check and report the use of sizes and alignments that the hardware cannot support.

The model implemented here does not mandate any specific data transfer mechanism. The APIs allow the user to install his own Interrupt Service Routine (ISR).

The TriMedia device libraries exist to create device drivers. Whereas device drivers are specific to an operating-system, the device libraries are generic. And whereas device drivers specify a data transfer mechanism, the device libraries leave the data transfer mechanism to the user.

An example shows how the SPDO device library can be used on its own without a traditional device-driver structure. An SPDO-based audio renderer that is a complete TSSA device driver will be available in the future.

## Using the SPDO API

The TriMedia SPDO API is contained in the archived device library libdev.a. To use the SPDO API, you must include the tmSPDO.h header files. The libdev.a device library is linked automatically.

**Note**
While developing programs using the device library, always use the debug version of the library (libdev_g.a). Numerous error conditions are trapped using asserts in the debug library.

The SPDO API depends upon the Board Support API, which is also included in libdev.a and is transparent to the user. If you want to change the hardware components on the board, see Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C.

## Limitations

You should be aware of the following hardware and/or software limitations:

- Calculation of the sample rate is based on TriMedia's cycle clock.The software gets its definition of this clock from a global variable that is patched when the program is loaded. Under Win95, this value is read from the tmman.ini file residing in the Windows directory (for Win95). It can be set explicitly using the debugger. You must ensure that the value specified matches your hardware.
- When setting sample rates, consider the fact that the value for the DDS control register is computed in 32-bit math, possibly resulting in inaccuracies because of truncation. The "GetSampleRate" functions return the actual sample as set in hardware.

## SPDO API Data Structures

This section presents the SPDO device library data structures. These data structures are defined in the tmSPDO.h header file.

| Name | Page |
|---|---|
| spdoCapabilities_t | 80 |
| spdoInstanceSetup_t | 82 |

## spdoCapabilities_t

```
typedef struct {
   tmVersion_t     version;
   Int             numSupportedInstances;
   Int             numCurrentInstances;
   char            codecName[DEVICE_NAME_LENGTH];
   UInt32          audioTypeFormats;
   UInt32          audioSubtypeFormats;
   UInt32          audioAdapters;
   Float           minSRate;
   Float           maxSRate;
   intInterrupt_t  intNumber;
   UInt32          mmioBase;
} spdoCapabilities_t, *pspdoCapabilities_t;
```

## Fields

| | |
|---|---|
| `version` | Version of the SPDO library module. Used by software to identify changes. |
| `numSupportedInstances` | Number of units in the hardware. |
| `numCurrentInstances` | Number of units currently in use. |
| `codecName` | A string giving the name of the codec installed (which varies depending on the board used). |
| `audioTypeFormats` | An OR'd value of all the audio type formats supported by this library. Audio type formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when spdoInstanceSetup is called. |
| `audioSubtypeFormats` | An OR'd value of all the audio subtype formats supported by this library. Audio subtype formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when spdoInstanceSetup is called. |
| `audioAdapters` | An OR'd value of all the audio adapters supported by this library. Audio subtype formats are defined in the file tmAvFormats.h. This will be updated with values from the board support package when spdoInstanceSetup is called. |
| `minSRate` | Lowest supported sample rate. |
| `maxSRate` | Highest supported sample rate. |
| `intNumber` | SPDO interrupt. |
| `mmioBase` | SPDO MMIO base address. |

### Description

A pointer to this structure is returned by the **spdoGetCapabilities** function. It is updated with values from the board support package when **spdoInstanceSetup** is called.

## spdoInstanceSetup_t

```
typedef struct {
   void                    (*isr)(void);
   intPriority_t             interruptPriority;
   Bool                      underrunEnable;
   Bool                      hbeEnable;
   Bool                      buf1emptyEnable;
   Bool                      buf2emptyEnable;
   tmAudioTypeFormat_t       audioTypeFormat;
   UInt32                    audioSubtypeFormat;
   tmAudioAnalogAdapter_t    output;
   Float                     sRate;
   Int                       size;
   Pointer                   base1;
   Pointer                   base2;
} spdoInstanceSetup_t, *pspdoInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `isr` | Interrupt service routine. |
| `interruptPriority` | The interrupt priority for the SPDO ISR installed. |
| `underrunEnable` | Enable interrupt sources. |
| `hbeEnable` | Enable HBE interrupt. |
| `buf1fullEnable` | Enable the ISR when buffer 1 is empty. |
| `buf2fullEnable` | Enable the ISR when buffer 2 is empty. |
| `audioTypeFormats` | The audio type format selected. Audio type formats are defined in the file tmAvFormats.h. |
| `audioSubtypeFormats` | The audio subtype format selected. Audio subtype formats are defined in the file tmAvFormats.h. |
| `output` | Selects the output, if multiple are available. The value **aaaNone** selects the default. |
| `srate` | Sample rate [Hertz]. |
| `size` | Size of buffers, in bytes, in samples. |
| `base1` | Base address of buffer 1. |
| `base2` | Base address of buffer 2. |

### Description

This struct is used by the function **spdoInstanceSetup** to read setup parameters. All fields are used on initial setup. Bases and size are updated only during the initial setup. The ISR, priority and flags can be updated while running.

## SPDO API Functions

This section presents the SPDO device library functions.

## spdoGetCapabilities

```
tmLibdevErr_t spdoGetCapabilities(
   pspdoCapabilities_t   *pCap
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return a pointer to capabilities data. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Returned if **pcap** is null. In the debug version of the library, an assertion triggers this when **pCap** is null. |

The function can also return codes from a call to **boardGetConfig**.

### Description

This function fills in the value of a user-supplied pointer variable which will then point to the single shared capabilities structure for the SPDO device library. The function is implemented by a call to **spdoGetCapabilitiesM** with **unitName** set to the first unit (**unit0**).

The first time this function is called, it retrieves the capabilities from the board support package.

## spdoGetCapabilitiesM

```
tmLibdevErr_t spdoGetCapabilitiesM(
   pspdoCapabilities_t   *pCap,
   unitSelect_t           unitName
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return a pointer to an SPDO capabilities structure. |
| `unitName` | Selects unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Unit not supported in hardware. |

### Description

The function finds the capabilities of the SPDO hardware.

## spdoGetNumberOfUnits

```
tmLibdevErr_t spdoGetNumberOfUnits(
   UInt32   *pNumberOfUnits
);
```

### Parameters

| | |
|---|---|
| pNumberOfUnits | Pointer to a variable in which to return the number of audio output units that are supported on the current hardware. |

### Return Codes

| | |
|---|---|
| TMLIBDEV_OK | Success. |
| TMLIBDEV_ERR_NULL_PARAMETER | In the debug version of the library, this assert is made if **pNumberOfUnits** is null. |

### Description

The function finds the number of audio output units that are supported on the current hardware.

## spdoOpen

```
tmLibdevErr_t spdoOpen(
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NO_MORE_INSTANCES` | Returned if there are no more free instances. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered if the instance is **NULL**. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Not supported in hardware. |

The function can also return codes from a call to **intOpen**. See the Interrupt API documentation for possible error codes.

### Description

This function opens an instance of the SPDO device. It opens the SPDO interrupt (intSPDO) with the **intOpen** function. The function is implemented by a call to **spdoOpenM** with **unitName** set to the first unit (**unit0**). This function is obsolete, and new code should use **spdoOpenM**.

## spdoOpenM

```
tmLibdevErr_t spdoOpenM(
   Int            *instance,
   unitSelect_t    unitName
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance variable. |
| `unitName` | Selects unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Not supported in hardware. |
| `TMLIBDEV_ERR_NO_MORE_INSTANCES` | Unit is already in use. |
| `TMLIBDEV_ERR_MEMALLOC_FAILED` | Memory used for instance variable structure. |

The function can also other return codes if allocation of the interrupt or hardware pins (on GPIO-enabled devices) fails.

### Description

This function will open an instance of the selected SPDO device and assign the instance value. It opens an interrupt appropriate to the specified unit using the **intOpen** function.

## spdoInstanceSetup

```
tmLibdevErr_t spdoInstanceSetup(
   Int                    instance,
   pspdoInstanceSetup_t   pSetup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value, assigned by **spdoOpen**. |
| `pSetup` | Pointer to setup structure. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | Returned if instance does not match owner. In the debug version of the library, this assertion is triggered instead of an error code. |
| `BOARD_ERR_NULL_FUNCTION` | Returned if the board codec structure is missing the initialization function (init_func) or the board codec structure is missing the setSRate function. |
| `AIO_INVALID_SIZE` | Can be asserted in debug mode if size is not a multiple of 64. |
| `AIO_INVALID_BASE` | Can be asserted in debug mode if base register is not cache aligned (multiple of 64 bytes). |

The function can also return codes from a call to boardGetConfig. Other error codes may be returned from a call to intInstanceSetup. See the interrupts API documentation for details. Other error codes can also be returned from the board codec initialization function (init_func) and the setSRate function. These functions are specific to the board codec.

### Description

This function performs initialization of the SPDO hardware by calling the appropriate init_func from the board API returned by the boardGetConfig function. It sets up the MMIO registers **SPDO_CTL**, **SPDO_BASE1** and **SPDO_BASE2** (**spdoSetBASE1** and **spdoSetBASE2** macros), and **SPDO_SIZE** (**spdoSetSize** macro). Then it sets up, when needed, the Interrupt Service Routine by calling the intInstanceSetup.

Upon return, the device is stopped but ready to go (by a call to **spdoStart**).

## spdoClose

```
tmLibdevErr_t spdoClose(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value, assigned in the call to **spdoOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | Returned if the instance does not match the owner. In the debug version of the library, an assertion triggers this when the instance does not match the owner. |

The function can also return codes from a call to intClose. See the Interrupts API document for possible codes. Other error codes can also be returned by a call to the board codec termination function. Those return values are specific to the board codec.

### Description

The function closes the instance of the SPDO device by calling the appropriate termination function from the board API, and close, using intClose, the interrupt opened by **spdoOpen**. After this function returns, the device is free and ready for re-allocation.

## spdoStop

```
tmLibdevErr_t spdoStop(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value, assigned in the call to **spdoOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | Returned if the instance does not match the owner. In the debug version of the library, an assertion triggers this when the instance does not match the owner. |

### Description

The function stops the audio play of this instance. This is achieved by calling the BSP's **stopFunc** function.

## spdoStart

```
tmLibdevErr_t spdoStart(
   Int    instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value, assigned in the call to **spdoOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | Returned if the instance does not match the owner. In the debug version of the library, an assertion triggers this when the instance does not match the owner. |
| `AIO_ERR_INIT_REQUIRED` | Returned if **spdoInstanceSetup** has not been called. |
| `BOARD_ERR_NULL_FUNCTION` | Returned if the board codec structure is missing the start function. |

The function can also return other error codes from the call to the board start function.

### Description

The function starts the audio play of this instance by calling the BSP's **startFunc** function.

## spdoSetSampleRate

```
tmLibdevErr_t spdoSetSampleRate(
   Int     instance,
   Float   srate
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value, assigned in the call to **spdoOpen**. |
| `srate` | Sample rate. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | Returned if the instance does not match the owner. In the debug version of the library, this assertion is triggered if the instance does not match the owner. |
| `AIO_ERR_SRATE_TOO_HIGH` | Returned if the requested sample rate is higher than the maximum allowed by the board codec chosen. |
| `AIO_ERR_SRATE_TOO_LOW` | Returned if the requested sample rate is lower than the minimum allowed by the board codec chosen. |
| `BOARD_ERR_NULL_FUNCTION` | Returned if the board codec structure is missing the setSRate function. |

The function can also return other codes from a call to the board codec setSRate function. Those codes are specific to the board codec.

### Description

The function sets or changes the sample rate, by calling the appropriate setRate function from the board API.

## spdoGetSampleRate

```
tmLibdevErr_t spdoGetSampleRate(
   Int     instance,
   Float   srate
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value, assigned in the call to **spdoOpen**. |
| `srate` | Pointer to a variable to hold the data. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | Returned if instance does not match the owner. In the debug version of the library, this assertion is triggered if instance does not match the owner |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assertion is triggered if **srate** is Null. |
| `BOARD_ERR_NULL_FUNCTION` | Returned if the board codec structure is missing the getSRate function. |

The function can also return codes from a call to the board codec getSRate function. Those codes are specific to the board codec.

### Description

The function gets the current sample rate, by calling the appropriate getSRate function from the board API.

## spdoConfig

```
tmLibdevErr_t spdoConfig(
   Int      instance,
   UInt32   subaddr,
   UInt32   value
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value, assigned in the call to **spdoOpen**. |
| `subaddr` | Generic pointer passed to codec Configure routine. |
| `value` | Generic pointer passed to codec Configure routine. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NOT_OWNER` | Returned if the instance does not match the owner. In the debug version of the library, an assertion triggers this when the instance does not match the owner. |
| `BOARD_ERR_NULL_FUNCTION` | Returned if the codec does not support Configure. |

The function can also return codes from the board codec ‘Configure’ function. Those codes are specific to the board codec.

### Description

The function allows the passing of configuration information from the user program through the device library to the (customized) user board codec routines. (Refer to the board API.) The two parameters, **subaddr** and **value**, are passed down to the board codec ‘Configure’ routine without any processing. The error code, if any, returned will be the return value of the board codec Configure routine.

### Implementation Notes

For boards provided by Philips, the board codecs do not provide a Configure function, and therefore, **spdoConfig** will return an error.

# Chapter 5

# Audio Digitizer (AdigAI) API

## Audio Digitizer API Overview

The audio digitizer provides a TSSA compatible interface to the audio input module. In addition to the expected dataout connection, the audio digitizer also provides an optional second output.

The audio digitizer is implemented with separate AL and OL layers. But no synchronous interface is provided. Hence, only the OL layer interface is documented.

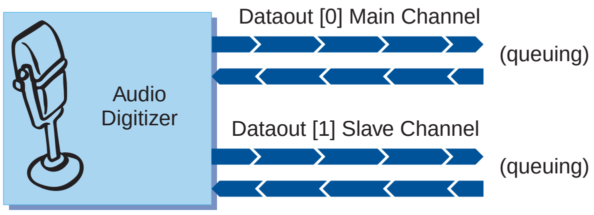


**Figure 1** Structure of the Audio Digitizer

While initial versions of the audio digitizer supported only the single audio in unit of the TriMedia processor, the current version of the audio digitizer has been updated in a compatible fashion to support the multiple audio in units that might be present on some TriMedia variants.

## Audio Digitizer Inputs and Outputs

The audio digitizer gets its capabilities from the underlying audio hardware. On the Philips IREF board, this is a stereo or mono 16-bit data stream. This capability can be altered in the audio input portion of the board support package. The Philips DTV reference board, for example, supports a choice between digital and analog inputs.

The digitizer has two output pins. These are referred to as the master and the slave. In normal operation, packets that are read from the empty queue on the master input are installed into the audio input hardware. When a given packet is full, an interrupt is triggered, and that packet is sent into the full queue.

When the slave output is enabled, a memory copy is performed inside the interrupt service routine so that the audio input stream can be routed to two independent modules.

## Audio Digitizer Errors

The audio digitizer supports the installation of and error callback function. This function will be called from the interrupt service routine, so make it brief. None of the errors handled by the error callback function are considered fatal. The error function prototype is of the type **tsaErrorFunc_t**:

```
typedef tmLibappErr_t(*tsaErrorFunc_t)(Int instId, UInt32 flags,
ptsaErrorArgs_t args);

typedef struct tsaErrorArgs {
   Int       errorCode;
   Pointer   description;
} tsaErrorArgs_t, *ptsaErrorArgs_t;
```

Handlers should be provided for these possible values of the errorCode:

`TMLIBAPP_ERR_OVERRUN`: The digitizer had samples available, but no memory was provided to store them. This buffer of samples has been dropped. The description field is a pointer to an integer array. Description[0] identifies the source of the error. Possible sources are:

0: Hardware overrun. Interrupts were locked out for too long.

1: Main channel overrun: No empty buffers available at main input.

2: Slave channel overrun. No empty buffers available at slave channel.

Description[1] contains the value of the CPU clock register when the error occurred.

`TMLIBAPP_ERR_HIGHWAY_BANDWIDTH_ERR`: The digitizer could not get access to the internal data "highway." Samples have been dropped. This situation might be corrected by changing the priority of the various DMA units. This is controlled using the **ARB_BW_CTL** MMIO register. See Chapter 19 of the appropriate TriMedia data book.

## Audio Digitizer Progress

You can optionally install a progress function to be notified when the format of the input stream has changed. Notification is required when the format of the input data is specified by an external master device, such as an SPDIF transmitter. The use of this mechanism is demonstated with the exolCopyAudio program running on a DTV reference board. When notified by an interrupt installed through the board support package, the digitizer progress function is called with the flag **AD_CHANGE_AT_DIGITAL_INPUT** when a change in format is detected.

## Audio Digitizer Configuration

The audio digitizer provides a simple configuration function. It can be used to change the input sample rate, change the source of the input stream, or to retrieve the status of the current input stream. Status is particularly useful when a digital audio input is being used. The exact behavior of these commands is determined by the board support package, as each command calls a function in the tmAI library.

```
tmLibappErr_t tmolAdigAIInstanceConfig (Int instance, UInt32 flags,
ptsaControlArgs_t args);
```

| | |
|---|---|
| `AD_SET_SAMPLE_RATE` | The parameter field of the control structure holds a floating point number that is used to set the sample rate, if possible. **aiSetSampleRate** is called. |
| `AD_GET_SAMPLE_RATE` | The parameter field of the control structure holds a pointer to a floating point number that is updated to reflect the current sample rate. **aiGetSampleRate** is called. |
| `AD_SET_VOLUME` | The parameter field of the control structure holds a pointer to an array of two integers that represents the left and right input gain in 0.01 dB steps. This is passed to the board support package the response is returned in the **retval** field of the arguments structure. **aiSetVolume** is called. |
| `AD_GET_VOLUME` | The last selected input volume is returned at the location to which the parameter field of the control structure points. The value is returned as an array of two integers specifying left and right input gain in 0.01 dB steps. **aiGetVolume** is called. |
| `AD_SET_INPUT` | The parameter field of the control structure holds an integer specifying the desired input source. Choices include: `aaaMicInput` `aaaLineInput` `aaaAuxInput1` `aaaAuxInput2` `aaaDigitalInput` as defined in the **tmAudioAnalogAdapter_t** enumerated in tmAvFormats.h. **aiSetInput** is called. |
| `AD_GET_INPUT` | The last selected input is returned at the location pointed to by the parameter field of the control structure. **aiGetInput** is called. |
| `AD_GET_FORMAT` | The format of the currently selected input is returned.This is particularly useful when locking to a digital input. **aiGetFormat** is called. |
| `AD_CONFIG` | The parameter field of the control structure is passed to the AI library's config function. This is used to implement other unspecified controls in the A/D converter. **aiConfig** is called. |

## Audio Digitizer API Data Structures

This section presents the Audio Digitizer application library data structures.

## tmolAdigAICapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
   Int32                       max_srate;
   Int32                       min_srate;
   Int32                       granularityOfAddress;
   Int32                       granularityOfSize;
   Int32                       minBufferSize;
   UInt32                      mmioBaseAddress;
} tmolAdigAICapabilities_t; *ptmolAdigAICapabilities_t;
```

### Fields

| | |
|---|---|
| `defaultCapabilities` | For compliance with the application library architecture, this is a pointer to a structure of the standard type. |
| `max_srate` | Minimum sample rate [Hz]. |
| `min_srate` | Maximum sample rate [Hz]. |
| `granularityOfAddress` | Number of lsb's that should be zero: (e.g. 6 for 64 byte alignment). |
| `granularityOfSize` | Number of LSBs that should be zero in the size field (size is the number of samples). |
| `minBufferSize` | Minimum buffer size (samples). |
| `mmioBaseAddress` | mmio base address of the selected audio out unit. |

### Description

The **tmolAdigAICapabilities_t** structure describes the capabilities and requirements of the audio digitizer module. A user can retrieve the structure's address by calling **tmolAdigAI-GetCapabilities**.

## tmolAdigAIInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t   defaultSetup;
   tmAdigAIMode_t               mode;
   Int32                        pauseBufferSize;
   tmAudioAnalogAdapter_t       input;
   Bool                         useControlIRQ;
} tmolAdigAIInstanceSetup_t; *ptmolAdigAIInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `defaultSetup` | For compliance with TSA, this is a pointer to a structure of the standard type. |
| `mode` | Can be **TMADIG_CONSERVATIVE_MODE** or **TMADIG_DIRECT_MODE**. Use only conservative mode. |
| `pauseBufferSize` | Specified in samples. A buffer of this size is allocated at instance setup. This buffer is used whenever overruns occur, as the digitizer needs to fill in the pointer so that capture does not overwrite other memory. |
| `tmAudioAnalogAdapter_t` | Can be line in, mic in, digital in, etc. |
| `input` | Input that will be used, the library will use the default input if input is set to aaaNone. |
| `useControlIRQ` | Indicates if the control IRQ should be used for digital audio input (if supported on the actual hardware). |

### Description

The **tmolAdigAIInstanceSetup_t** structure describes the intended operation of this instance of the digitizer.

## Audio Digitizer API Functions

This section presents the Audio Digitizer API application library functions.

| Name | Page |
|---|---|
| tmolAdigAIGetCapabilities | 105 |
| tmolAdigAIGetCapabilitiesM | 106 |
| tmolAdigAIGetNumberOfUnits | 107 |
| tmolAdigAIOpen | 108 |
| tmolAdigAIOpenM | 109 |
| tmolAdigAIGetInstanceSetup | 110 |
| tmolAdigAIInstanceSetup | 111 |
| tmolAdigAIStart | 112 |
| tmolAdigAIStop | 113 |
| tmolAdigAIInstanceConfig | 114 |

## tmolAdigAIGetCapabilities

```
tmLibappErr_t tmolAdigAIGetCapabilities(
   ptmolAdigAICapabilities_t   *pCap
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return a pointer to capabilities data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Used to retrieve the capabilites of the audio digitizer. The function pointer that is returned remains valid as long as the digitizer is active. Implemented by a call to **tmol-GetCapabilitiesM** with **unitName** set to the first unit (**unit0**).

## tmalAdigAIGetCapabilitiesM

```
tmLibappErr_t tmalAdigAIGetCapabilitiesM(
   ptmalAdigAICapabilities_t   *pCap,
   unitSelect_t                 unitNumber
);
```

## tmolAdigAIGetCapabilitiesM

```
tmLibappErr_t tmolAdigAIGetCapabilitiesM(
   ptmolAdigAICapabilities_t   *pCap,
   unitSelect_t                 unitNumber
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to pointer to an AdigAI capabilities structure at the appropriate level (AL or OL). |
| `unitName` | Select which unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Unit not supported in hardware. |

### Description

Used to find out about the AI hardware.

## tmalAdigAIGetNumberOfUnits

```
tmLibappErr_t tmalAdigAIGetNumberOfUnits(
   UInt32   *numberOfUnits
);
```

## tmolAdigAIGetNumberOfUnits

```
tmLibappErr_t tmolAdigAIGetNumberOfUnits(
   UInt32   *numberOfUnits);
```

### Parameters

| | |
|---|---|
| `numberOfUnits` | Pointer to a variable in which to return the number of audio input units that are supported on the current hardware. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assert is given if **numberOfUnits** is null. |

### Description

Used to find the number of audio input units that are supported on the current hardware.

## tmalAdigAIOpen

```
tmLibappErr_t tmalAdigAIOpen(
   Int   *instance
);
```

## tmolAdigAIOpen

```
tmLibappErr_t tmolAdigAIOpen(
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Address of an integer that will hold the instance value for this audio digitizer |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | No more instances of the audio digitizer are available. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Memory allocation for the default instance variables failed. |

### Description

The open function creates an instance of the audio digitizer and informs the user of its instance. The audio digitizer supports only one instance. Add to description: Implemented by a call to **tmolOpenM** with **unitName** set to the first unit (**unit0**).

## tmalAdigAIOpenM

```
tmLibappErr_t tmalAdigAIOpenM(
   Int           *instance,
   unitSelect_t   unitNumber);
```

## tmolAdigAIOpenM

```
tmLibappErr_t tmolAdigAIOpenM(
   Int           *instance,
   unitSelect_t   unitNumber
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance. |
| `unitName` | Select unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Unit not supported in hardware. |
| `TMLIBDEV_ERR_NO_MORE_INSTANCES` | Unit is already in use at the device library level. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Unit is in use at the AL or OL level. |
| `TMLIBDEV_ERR_MEMALLOC_FAILED` | Memory used for instance variable structure. |

Other errors might be returned if the allocation of the interrupt or hardware pins (on GPIO enabled devices) fail.

### Description

This function will open an instance of the selected AI device and assign the instance value. Using the tmAI device library, It opens an interrupt as appropriate for the specified unit with intOpen function.

## tmolAdigAIGetInstanceSetup

```
tmLibappErr_t tmolAdigAIGetInstanceSetup(
   Int                         instance,
   ptmolAdigAIInstanceSetup_t  *setup
);
```

### Parameters

| | |
|---|---|
| `instance` | As returned from **tmolAdigAIOpen**. |
| setup | Pointer to variable in which to return a pointer to setup data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

The functio retrieves a pointer to the current instance setup. After a call to **tmolAdigAI-Open**, this structure is filled with default values to simplify the impending call to **tmol-AdigAIInstanceSetup**.

## tmolAdigAIInstanceSetup

```
tmLibappErr_t tmolAdigAIInstanceSetup(
   Int                          instance,
   ptmolAdigAIInstanceSetup_t   setup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmolAdigAIOpen**. |
| `setup` | Pointer to setup data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Specified instance does not match current instance. Digitizer supports only one instance. |
| `TMLIBAPP_ERR_NULL_DATAOUTFUNC` | A valid dataout function is required. Only streaming operation is supported. |
| `TMLIBAPP_ERR_INVALID_SETUP` | The primary output is not enabled. Queues or format are unspecified. The primary output must be connected. |
| `TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE` | An unsupported data format was requested. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Memory allocation for the pause buffer failed. |

Other errors are possibly reported by the device library or board support package.

### Description

The audio digitizer is prepared for operation. Parameters are checked. The digitizer is left “stopped.” It will become operational on a call to **tmolAdigAIStart**. The interrupt service routine is running. Data is being captured and thrown away in the ‘pause’ buffer.

## tmolAdigAIStart

```
tmLibappErr_t tmolAdigAIStart(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmolAdigAIOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

The digitizer represented by the instance is started. This causes data pointers from **tmAvPacket_t** packets to be installed into the audio hardware so that the audio DMA engine can capture data for delivery to the application.

## tmolAdigAIStop

```
tmLibappErr_t tmolAdigAIStop(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned **tmolAdigAIOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

The digitizer represented by the instance is stopped. This causes the pause buffer to be installed into the registers of the AI DMA engine. The interrupt stays active until **tmolAdigAIClose** is called.

## tmolAdigAIInstanceConfig

```
tmLibappErr_t tmolAdigAIInstanceConfig(
   Int                instance,
   UInt32             flags,
   ptsaControlArgs_t  args
);
```

### Parameters

| | |
|---|---|
| `instance` | As returned from **tmolAdigAIOpen**. |
| `flags` | Not used by **tmolAdigAIInstanceConfig**. |
| `args` | Points to a control structure used to modify the operation of the audio digitizer. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

Errors detected by the underlying **tmalAdigAI** call can be found in the **retval** member of the control structure.

### Description

The parameters of the configuration function are described on page 100 (in this chapter).

Chapter 6

# Audio Renderer (ArendAO) API

## Audio Renderer API Overview

The Audio Renderer for TriMedia serves as a TSSA-compatible interface between audio stream-producing modules and the outside world. The renderer receives packets of data and, using the AO hardware, converts it to analog audio.

The audio renderer supports both streaming and non-streaming interfaces to the audio hardware. To provide these services, the renderer installs an interrupt service routine. The renderer supports a sophisticated form of synchronization based on presentation times and a reference clock. The audio renderer is supplied as a library that can be used without restriction by owners of the TriMedia SDE. For those that are interested, similar audio interrupt service routines are presented in the programs that are included in the TCS examples directory. Refer in particular to patest.c.

Like all TSSA components, the renderer requires audio data to be packaged in **tmAvPacket_t** data structures. (For more information, see Chapter 4, *tmAvFormats.h: Multimedia Format Definitions* in Book 3, *Software Architecture*, Part A.)

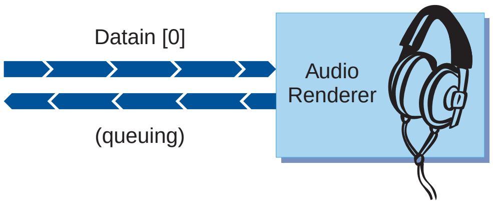


**Figure 2** Structure of the Audio Renderer

While initial versions of the audio renderer supported only the single audio out unit of the TriMedia processor, the current version of the audio renderer has been updated in a compatible fashion to support the multiple audio out units that might be present on some TriMedia variants.

### Inputs and Outputs

The Audio Renderer has one input and consumes buffers full of audio data. It returns the same buffers in an empty state. In exchange, sound is produced. The exact mechanism depends on the available hardware. The audio renderer is customized for new boards using the mechanism of the board support package.

### Errors

Errors can be reported during the renderer's setup phase, or at run time. Errors reported during the setup phase will be noticed as non-zero return values from the API. In addition, the library used in its debugging mode will use the assert mechanism to flag invalid inputs. These errors are covered along with the descriptions of each function in the API.

**Note**
We strongly advise that you bring up the audio renderer using the _a (assert) version of the audio renderer and the device libraries. Many possible error conditions are flagged with assertions in these libraries.

For run-time errors, the audio renderer supports an error-reporting callback function. This function will be called from the audio output interrupt service routine. None of the errors handled by the error callback function are considered fatal. The error function prototype is of the type **tsaErrorFunc_t**:

```
typedef tmLibappErr_t(*tsaErrorFunc_t)(Int instId,
   UInt32 flags, ptsaErrorArgs_t args);
```

```
typedef struct tsaErrorArgs {
   Int           errorCode;
   Pointer       description;
} tsaErrorArgs_t, *ptsaErrorArgs_t;
```

Handlers should be provided for these possible values of the error code:

## TMLIBAPP_ERR_UNDERRUN

The audio system requested data but none was available. This error could come up in several circumstances. The renderer's interrupt service routine always attempts to handle this case gracefully. More information about the source of the data is available in the description field. When an underrun error is logged, the renderer fills in the description field with a pointer to an array of three integers. The first member identifies the exact source of the error.

```
Values of errArg.description[0]:
0: The interrupt handler was locked out for too long.
1: The streaming handler found the queue empty.
```

## TMLIBAPP_ERR_HIGHWAY_BANDWIDTH_ERR

The receipt of this error implies that there is not enough bandwidth available to service audio output on the TriMedia's internal data highway. This can be remedied by reprogramming the bandwidth allocation MMIO register.

## AR_ERR_BUF_TOO_LARGE

This message is only given by the streaming interrupt service routine. When the audio renderer operates in streaming mode, it allocates a buffer of silence. This is played whenever valid audio data is not available. As a result, the largest buffer size needs to be specified when the renderer is setup.

**Note**
Since the error function is called from the context of an interrupt, it is not appropriate to call **printf** or any other complex handler.

## Progress Function

When the renderer is operated in streaming mode, a progress function can be called according to the user's needs. If this is not desired, set the **progressFunc** member of the instance setup structure to Null. The **progressReportFlags** member of the default instance setup structure allows a user to control when the progress function is called. The valid flags, defined in tmalArendAO.h, include **AREND_PROGRESS_ReportCount** and **AREND_PROGRESS_SyncEventCorrect**. The **ReportCount** flag causes the progress function to be called in every ISR. The **SyncEventCorrect** flag causes the progress function to be called when the AV sync algorithm is active. The other values of this flag are reserved.

The progress function has this prototype:

```
typedef tmLibappErr_t (*tsaProgressFunc_t) (Int instId, UInt32 flags,
   ptsaProgressArgs_t args);
```

```
typedef struct tsaProgressArgs {
   Int           progressCode;
   Pointer       description;
} tsaProgressArgs_t, *ptsaProgressArgs_t;
```

When called, the progress code will be some combination of the progress flags, as noted above. It is possible for the progress function to be called with more than one flag set. In the audio renderer, the description field of the **args** structure will usually contain a pointer to a structure of this type:

```
typedef struct tmArendAOControlInfo {
   UInt             numberOfSamples;
   ptmAvPacket_t    packet;
   Int              timeDiff;
   Int              muteTimer;
   arSyncState      syncState;
} tmArendAOControlInfo_t, *ptmArendAOControlInfo_t;
```

One exception is made for the progress flag **AREND_PROGRESS_ChangeSampleRate**. In that case, the parameter member points to a floating point sample rate. In the control info structure, the **numberOfSamples** field gives the number of samples that will be played when the buffer installed by this interrupt is played. Knowledge of this number allows a user to implement a simple form of synchronization using the mechanism of the DDS clock synthesizers available on TriMedia. For example, the output clock can be phase locked to an external source at the audio input. The packet member of this structure is the address of the audio packet that is being installed in this interrupt. The **timeDiff** member of the structure is used to implement the more sophisticated type of synchronization appropriate for time stamped data streams. The **syncState** and **muteTimer** are used to monitor the operation of the internal synchronization mechanism. They are described in the API reference entry for this structure.

## How to Use Audio Renderer

The audio renderer can be used in streaming or non-streaming mode. In streaming mode, it can also be used at the AL or OL layer. These terms are explained in full in Book

3, *Software Architecture*. Two examples are provided explicitly to illustrate the use of the audio renderer. **exalArendAO** demonstrates both streaming and non-streaming operation without an operating system at the AL layer. **exolArendAO** demonstrates streaming operation at the OL layer, using the TSSA default functions and pSOS.

The AL layer of the audio renderer is appropriate for use in situations where an operating system is not required. It is also used to implement an operating system layer (OL). The AL layer supports a function-based interface (non-streaming), as well as a streaming interface, via callbacks.

1. Call the Open function so that an instance can be assigned. The audio renderer will support as many audio outputs as are available on the processor. For 32-bit TriMedia processors, this is one.
2. Obtain a pointer to an instance setup structure and fill it in. This structure completely describes the operation of the renderer. Most of the important fields are in the "standard" location. Refer to tsa.h for the definition of the structure **ptsaDefaultInstanceSetup_t**. In particular, you will fill in the format structure. Entries are provided for a number of callback functions. When working at the OL layer, the datain function is provided as a default. The presence of the datain function is used to determine whether the renderer runs in streaming, or non-streaming mode. Finally, call **tmalArendAOInstanceSetup**.
3. In streaming mode, call the Start function. This will cause the renderer to expect data and consequently, to log ensuing errors if data is not present.
4. In non-streaming mode, call **tmalArendAORenderBuffer** to send audio data to the DAC. You can use the completion function or you can poll the **buffersInUse** field of the packet that you sent to determine when the buffer has been rendered. In non-streaming mode, the application calls a library function to send data downstream. In the audio renderer, this is **tmalArendAORenderBuffer**. By contrast, in streaming mode the application uses a queuing mechanism (provided by the operating system) to send data. The timing of the data flow is determined by the availability of empty buffers, rather than by the availability of buffers full of data. The operation of the renderer also changes accordingly. For example:

```
Non-Streaming Mode:
The waiting program spins waiting for data.
   while( 1 ){
      computeData();
      render frame();
   while (buffer not available);
   }
Streaming Mode:
The waiting program gives up control to the OS via the "q_get" call.
   while( 1 ){
      q_get (BLOCK);
      computeData();
      q_put();
   }
```

## How the Audio Renderer Works

The audio renderer installs an interrupt service routine and all processing happens in this ISR. This version of the audio renderer uses the audio out (AO) device library to render sound. Other versions of an audio renderer might use the VO or SSI hardware interfaces. Since the API is identical, an application using the OL layer to produce audio wouldn't know the difference.

The OL layer always runs in streaming mode, and this is the more sophisticated method of operation. All setup happens in the **(tmxx)ArendAOInstanceSetup** function. An interrupt service routine is installed. If no **datainFunc** is installed, we are in non-streaming ("pull") mode, and a simple queue is initialized. The queue is written to in the **tmalArendAORenderBuffer** function. It is read during the interrupt service routine.

A more complete description of the use of the audio renderer in streaming mode is found in the example program "exolArendAO." This is also documented in Chapter 6, "Programming TriMedia Audio Applications," of the *Cookbook*. As with any OL layer component constructed to the streaming architecture, the setup happens through the structures passed to **tmolArendAOInstanceSetup**. Communication queues are allocated and buffers are placed in the empty queue. As the source component is initialized with the same pair of queues, data exchange begins as soon as **tmolArendAOStart** is called.

When streaming mode is selected by the installation of a datain function, the **datainFunc** will be called in the interrupt service routine. This allows you to install your own (operating system based) queueing system.

### The Silence Buffer

The **maxBufferSize** field of the instance setup variable is used to allocate a "silence buffer." This buffer is zeroed, and it is played whenever valid data is not available.

### Raw Mode and Conservative Mode

In Conservative mode, the audio renderer copies back all buffers before sending them to the audio hardware. This relieves the user of cache coherency responsibility. In Raw mode, it is up to the user to ensure cache coherency. When the TSSA default OL layer interface is used, cache coherency is handled by the default functions.

## Formats in the Audio Renderer

The list of formats supported by the audio renderer is determined by a call to **tmAOGetCapabilities**. This in turn queries the board support package. The audio renderer itself supports all 16- and 32-bit formats between mono and eight channels. Which of these are available is up to the board support package.

The OL version of the audio renderer can change its format on-the-fly. It is possible (and legal) to specify no format at instance setup, instead relying on the format to be passed in the first data packet. Similarly, the audio renderer will change its format based on the format specified in any packet. Because the format change function cannot be called from the interrupt service routine, it is called from the sending component's dataout function. This may result in a few queued packets being played with the wrong format.

## Synchronization Overview

The audio renderer includes a number of services designed to be used for synchronization. It also includes specific code to synchronize audio and video streams, based on a reference clock and time stamped packets. The AV sync code is used, for example, in the TriMedia DTV reference application. Other forms of sync, such as "broadcast sync" and "AA sync" are supported with the user's ability to change the audio sample rate and the reports given by the renderer in the progress function. The interface includes a **syncMode** parameter. When this is set to **AR_Sync_None**, sync is disabled and all packets are played as received. In **AR_Sync_trigger** mode, the renderer expects the first packet it sees to be time stamped, and this packet is held until the reference clock is greater than the time stamp. The **AR_Sync_skip** mode is the most powerful.

### AV Sync Details

There are many details involved in the renderer's AV synchronization scheme. When the **AR_Sync_trigger** mode is used, the operation is almost trivial. An example of the complete **AR_Sync_skip** mechanism is demonstrated in the DTV reference app, ATSCbasic. The mechanism is still conceptually simple, as described below:

In order for the AV sync mechanism to be enabled, the user must explicitly enable the mechanism by setting the syncMode to **AR_Sync_skip**. Also, a reference clock must be installed at instance setup, and packets to be synchronized must contain a valid timestamp. With these pre-requisites met, the algorithm attempts to present the packet at the correct time. This behavior is controlled by the user's specification of the **syncMode** and the **timeThreshold**. In very general terms:

1. If the time stamp matches the clock to the accuracy of the **timeThreshold** field, the packet will be played. The user is informed of this condition through the progress function, and the **syncState** value of **AR_SyncState_CorrectionAppropriate** informs him that this is a good time to fine tune the audio sample clock, for example, with the DDS.
2. If the time stamp is earlier than the current clock value, the packet is returned without being fully played.
3. If the time stamp is in the future, the packet is held until the clock reaches the value of the timestamp
4. If the difference between the time stamp and the clock is greater than 16 times the **timeThreshold**, then the time stamp data is assumed to be erroneous and it is ignored. Note that it is this parameter that determines the number of packets that must be

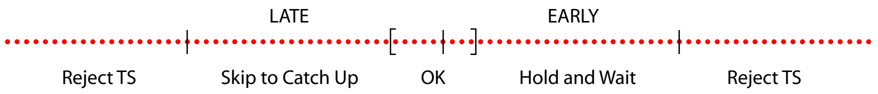
available in the system. If the audio renderer can hold a packet for 16 times the threshold, the rest of the system must be prepared for this possibility.

### timeDiff

The fundamental measure of AV sync is the comparison between the Presentation Time Stamp (PTS) and the clock, sometimes called the Program Clock Reference (PCR). This comparison is made using the difference (**timeDiff**) between these two clocks, each represented by a 32-bit number. When the difference is positive, the packets are ahead of the clock. Since humans prefer late audio to early audio, the skipping algorithm is biased to return slightly negative time differences.

### timeThreshold

The **timeThreshold** member of the instance setup structure sets how far out of sync the clock and the PTS can be before drastic action is taken. When the difference (**timeDiff**) is less than the threshold, the audio can track the video without gaps in the audio data. In MPEG applications, the PCR usually runs at 90KHz, and a threshold of 3000 works well.



When a timestamp is encountered that is inside of the rejection window, but outside of the acceptable window, and early, it is held and silence is played until the timestamp matches the reference clock. If it is late, a short packet of silence is played (64 samples) and, the next packet is retrieved.

As a further conservative measure, timestamps must indicate the necessity of skipping or waiting three times consecutively before any action is taken.

As mentioned, the human bias against early audio causes the threshold to be asymmetrical around zero. A value of timeThreshold/2 is used for positive time differences.

### Rejecting Bad Time Stamps

When the absolute value of the time difference is greater than sixteen times the threshold, it is assumed that the clock or the time stamp is bad, and the packet is played as if it were not time stamped.

### Holding a Packet When Ahead

When the time difference is greater than the half threshold (but less than 16 times the threshold), the packet is ahead of the clock and the renderer will hold this packet until the clock catches up. The progress function is called with the **syncState** set to **AR_SyncState_Waiting**. When a packet is being held, the audio renderer output is muted, the mute counter is initialized to the value of **muteCounterInit** as specified in the instance setup structure, and silence is played. This mechanism places a minimum on the number of packets that are available in a system. If the threshold is set high, and too few packets are circulated in the queues, it may be possible for all of the packets to end up held by the audio system because of this case. The threshold should be set appropriately and enough packets should be available so this does not become an issue.

### Playing Short to Catch Up

When the time difference is less than the negative threshold (but not less than 16 times the negative threshold), the packet is behind the clock, and the audio needs to catch up. The renderer catches up by playing only the minumum number of samples from the packet. This number is 64 stereo 16-bit samples, but it is smaller for multi-word samples. Note that it might be possible to break this mechanism if you routinely use very short buffers with the renderer. The progress function is called with the **syncState** set to **AR_SyncState_Skipping**. Like in the “holding” case, the audio renderer output is muted, the mute counter is initialized to the value of **muteCounterInit** as specified in the instance setup structure, and silence is played while the renderer is skipping.

### Adapting the Sample Rate

When the time difference is within the bounds set by the threshold, the application is given the opportunity to drive the remaining difference to zero using some sort of a linear controller in a feedback loop. The controller can be very simple: If the output clock is based on the TriMedia DDS, the output sample rate linearly follows the DDS control word. The DDS control word can be modified according to an equation like this:

$$\text{newDDS} = (1 - \text{Kp} \times \text{timeDiff}) \times \text{originalDDS}.$$

This sort of an update is normally done when the progress function is called with **syncState** equal to **AR_SyncState_CorrectionAppropriate**. For more information, see the example code in the ATSCbasic application.

### syncDelay

It is often useful to add an offset to the PCR (clock) when computing the time difference. The audio renderer can accept this offset as the **syncDelay**. It can be specified at instance setup, or changed on the fly using the **AR_SET_DELAY** command to the **InstanceConfig** function. The **syncDelay** is added to the **timeDiff**, so a positive delay will move the packet forward in time.

As an example, the **syncDelay** is useful when the video renderer is also locking to the same PCR. The video renderer can only lock to an accuracy of one frame, but it can measure an offset to the clock of less than one frame. In the DTV applications, the video renderer passes this offset to the audio renderer as a the syncDelay, and the audio renderer uses this to compute the time difference.

But when the delay tends to jump abruptly, it might be appropriate to filter the delay so that artifacts in the sound are less noticeable. To do this, do not use the audio renderer’s delay parameter. Instead, add the filtered delay value to the time difference reported in the progress function. A filter like this will be updated only when a packet with a time-stamp is processed.

### Muting

Whenever the renderer is muted, a counter internal to the renderer is initialized to the value specified as **muteCounterInit** at Instance Setup. When mute is disabled, the counter begins to count down and mute is actually lifted when the counter gets back to zero. If this feature is not desired, simply set **muteCounterInit** to zero. But the mute counter is also used when muting is enabled by the AV sync mechanism. The counter is used to avoid the situation where several short periods of mute are heard while the AV sync algorithm locks up. It is better to stay muted until everything is stable.

## Other Forms of Sync

Another aspect of synchronization occurs when the audio output should be slaved to the audio input. This occurs when an AO input is being used. This form of sync is referred to as AA sync, for Audio-Audio. The mechanism for AA sync is generally implemented outside of the audio renderer through the use of the callback function. AA sync is likely to be incompatible with AV sync, as only one master is reasonably allowed. The TriMedia DTV Audio System module includes an implementation of AA sync. The concepts are fairly simple. The DDS value is scaled in accordance with the difference between the number of incoming samples and outgoing samples.

Broadcast sync is required when the system is a slave to a signal that is being broadcast. Some mechanism has to lock the receiver's clocks to those of the broadcaster. The audio renderer's **MODIFY_SRATE** command can be used to implement this sync. Or this requirement can be met through the lock of time stamps to the PCR when the PCR is locked to the broadcast stream.

## Audio Renderer API Data Structures

This section presents the Audio Renderer application library data structures.

| Name | Page |
|---|---|
| arMode_t | 126 |
| arConfigParam_t | 127 |
| arProgressFlags_t | 128 |
| arSyncMode_t | 129 |
| arSyncState_t | 130 |
| tmalArendAOCapabilities_t | 131 |
| tmolArendAOCapabilities_t | 131 |
| tmalArendAOInstanceSetup_t | 132 |
| tmolArendAOInstanceSetup_t | 132 |
| tmArendAOControlInfo_t | 134 |

## arMode_t

```
typedef enum {
   AR_MODE_RAW,
   AR_MODE_CONSERVATIVE
} arMode_t;
```

### Description

The renderer can be run in raw or conservative mode. These are legal values to be passed to **tmalArendAOInstanceSetup**. In conservative mode, audio data to be rendered is copied back from cache to SDRAM before playback. In raw mode, this operation is the responsibility of the user. The TSA defaults handle this, and the OL version of the renderer runs in RAW mode.

## arConfigParam_t

```
typedef enum {
   AR_VOLUME,
   AR_PAN,
   AR_SAMPLE_RATE,
   AR_MUTE,                     /* toggle */
   AR_SET_MUTE,
   AR_GET_MUTE,
   AR_SET_DELAY,
   AR_SET_SAMPLE_RATE,
   AR_GET_SAMPLE_RATE,
   AR_SET_SYNC_MODE,
   AR_GET_SYNC_MODE,
   AR_SET_BAD_TIMESTAMP_THRESHOLD,
   AR_SET_SYNC_THRESHOLD,
} arConfigParam_t;
```

### Description

Describes the quantities that can be adjusted using the instance config functions. Volume and pan are passed as integers. The requested value is passed through the device library and to the board support package where the command may or may not be supported. Both values are specified in "milliBels" (1/100th of a DB). Positive pan values go right.

Sample rate is passed as a pointer to a floating point number. The obsolete **SR_SAMPLE_RATE** command is identical to **AR_SET_SAMPLE_RATE**.

The **MUTE** command toggles the mute state. When **SET_MUTE** is used, the requested value is passed directly as the parameter. When **GET_MUTE** is used, the user passes the address of a Boolean variable.

**DELAY** is specified in units of the currently installed TSA clock, and it is passed directly.

When **SET** or **GET_SYNC_MODE** are used, the address of the mode is passed as the parameter.

When syncSkip mode is active, packets can be rejected if their timestamp is too far from the current clock value. The **AR_SET_BAD_TIMESTAMP_THRESHOLD** configuration command allows the threshold of rejection to be changed dynamically. Default value is 48000 ticks.

The **AR_SET_SYNC_THRESHOLD** configuration command dynamically adjusts the difference between the reference clock and a timestamp on a packet that is considered "close enough" for playback.

## arProgressFlags_t

```
typedef enum {
   AREND_PROGRESS_ReportCount,
   AREND_PROGRESS_ChangeSampleRate,
   AREND_PROGRESS_SyncEventCorrect,
   AREND_PROGRESS_EndOfStream,
   AREND_PROGRESS_ChangeFormat
} arProgressFlags_t;
```

### Description

The renderer can call the progress function under a number of conditions. **EndOfStream** and **ChangeFormat** are TSSA standard progress occasions.

The **ReportCount** flag causes the progress function to be called at every interrupt service routine to report the count of samples played, using the **ControlInfo** type described below.

The **ChangeSampleRate** flag causes the progress function to be called if the sample rate is changed. This gives an opportunity for the application's sync algorithm to be informed of sample rate changes. Note that the progress function is not called for the initial installation of the sample rate, as the sample rate is set before the progress function variable is set.

The **SyncEventCorrect** flag causes the progress function to be called if a timestamp was detected in a received packet, and hence a sync event was triggered. This progress event is critical to the implementation of AV sync algorithms. When called, the progress parameter is a pointer to a **ControlInfo** structure, as described below.

## arSyncMode_t

```
typedef enum {
   AR_Sync_None,
   AR_Sync_trigger,
   AR_Sync_skip,
} arSyncMode_t;
```

### Description

Synchronization processing can be disabled (**AR_Sync_None**), or it can be enabled in one of two modes. In “trigger” mode, the renderer expects the first packet it receives to be time stamped, and this packet is held until the reference clock matches the time stamp. After that, sync information is ignored, and all packets are played. The trigger is reset by stopping and starting the renderer.

In “skip” mode, the renderer uses an algorithm as described earlier, under AV Sync Details. Packets are constantly checked for their relation to the reference clock, and they are always presented within the window specified.

## arSyncState_t

```
typedef enum {
   AR_SyncState_NoAction,
   AR_SyncState_Waiting,
   AR_SyncState_Skipping,
   AR_SyncState_CorrectionAppropriate,
} arSyncState_t;
```

### Description

When synchronization processing is enabled, the **syncState** is communicated to the application using the controlInfo structure as passed in the progress function. The skipping and waiting states tell the application that the timestamp of the packet to be played is outside of the legal window. When the **CorrectionAppropriate** state is used, the packet is within its window and further action is up to the application. It is “appropriate” to “correct” the sample rate clock.

## tmalArendAOCapabilities_t

```
typedef struct tmalArendAOCapabilities_t {
   ptsaDefaultCapabilities_t   defaultCapabilities;
   Int32   max_srate;
   Int32   min_srate;
   Int32   granularityOfAddress;
   Int32   granularityOfSize;
   Int32   minBufferSize;
} tmalArendAOCapabilities_t; *ptmalArendAOCapabilities_t;
```

## tmolArendAOCapabilities_t

```
typedef tmalArendAOCapabilities_t
tmolArendAOCapabilities_t,  *ptmolArendAOCapabilities_t;
```

### Fields

| | |
|---|---|
| `defaultCapabilities` | For compliance with the application library architecture, this is a pointer to a structure of the standard type. |
| `max_srate` | Minimum sample rate [Hz]. |
| `min_srate` | Maximum sample rate [Hz]. |
| `granularityOfAddress` | Number of LSBs that should be zero (for example, 6 for 64-byte alignment). |
| `granularityOfSize` | Number of LSBs that should be zero in the size field (size is the number of samples). |
| `minBufferSize` | Minimum buffer size (samples). |

### Description

**tmalArendAOCapabilities_t** and **tmolArendAOCapabilities_t** are structures holding a list of capabilities. The audio renderer maintains a structure of this type to describe itself. A user can retrieve the structure's address by calling **tmalArendAOGetCapabilities** or **tmolArendAOGetCapabilities**. Notice that the AL and the OL layer structures are identical, except for the extensions to the default capabilities structure made in the OL layer (tsa.h)

## tmalArendAOInstanceSetup_t

```
typedef struct tmalArendAOInstanceSetup_t {
   ptsaDefaultInstanceSetup_t   defaultSetup;
   arMode_t                     operationalMode
   Int32                        maxBufferSize;
   tmAudioAnalogAdapter_t       output;
   Int                          muteCounterInit;
   UInt32                       syncThreshold;
   Int                          syncDelay;
   arSyncMode_t                 syncMode;
   Int32                        badTimestampThreshold;
} tmalArendAOInstanceSetup_t;
```

## tmolArendAOInstanceSetup_t

```
typedef tmalArendAOInstanceSetup_t
tmolArendAOInstanceSetup_t, *ptmolArendAOInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `defaultSetup` | Refer to tsa.h for more information. The function pointers (error func, datain func) are taken from here, as is the format. |
| `operationalMode` | Raw or Conservative. See **arMode_t** on page 126. |
| `maxBufferSize` | Maximum buffer size in bytes. The value of this field should match the buffer size used in the data packets that are played by the renderer. Because of hardware restrictions, the number of samples must be a multiple of 64d. **maxBufferSize** must hence be a multiple of 64, and the number of bytes per sample in the chosen data format (for example, 12 for six channel 16-bit). This size is used to allocate a “silence buffer” that is played whenever appropriate. |
| `output` | Select line output or digital output. The value **aaaNone** selects the default. |
| `muteCounterInit` | Whenever the audio renderer is muted, either by user command or by loss of AV sync, a counter internal to the renderer is initialized to this value. It is then decremented when the mute condition is lifted. Only when the counter goes to zero is the mute actually ended. Set to zero to disable this feature. |

| | |
|---|---|
| `syncThreshold` | When the difference between the current time and the timestamp on a packet exceeds this threshold, packets are held or skipped to correct the loss of sync. This mechanism is activated if 1) A clock is installed at instance setup. 2) Valid timestamps are provided on received packets. A value equal to 30ms is usually appropriate. |
| `syncDelay` | An offset, given in units of the installed TSA clock that is used to compute audio video (AV) sync. Positive values delay the playback of packets. See page 123. |
| `syncMode` | See **arSyncMode_t** enum above. |
| `badTimestampThreshold` | When **syncSkip** mode is active, packets can be rejected if their timestamp is too far from the current clock value. This configuration command allows the threshold of rejection to be changed dynamically. Default value is 48000 ticks. |

## Description

A structure of this type is passed to **tmalArendAOInstanceSetup** or to **tmolArendAO-InstanceSetup.**Using the standard tmal (TriMedia Application Library) model, you can configure the renderer at the AL or OL layer. The OL layer simply calls the AL layer.

## tmArendAOControlInfo_t

```
typedef struct tmArendAOControlInfo {
   UInt            numberOfSamples;
   ptmAvPacket_t   packet
   Int             timeDiff;
   Int             muteTimer;
   arSyncState_t   syncState;
} tmArendAOControlInfo_t, *ptmArendAOControlInfo_t;
```

### Fields

| | |
|---|---|
| `numberOfSamples` | The number of samples that will be played in this invocation of the audio renderer ISR. |
| `packet` | Pointer to the packet that will be played in this invocation of the audio renderer ISR. |
| `timeDiff` | Difference in time, give in the units of the currently installed TSA clock, between the time stamp of the current packet, and its expected presentation time, with the addition of the syncDelay. See page 122. |
| `muteTimer` | The current value of the mute timer. |
| `syncState` | Tells the progress function what decision the renderer has made about sync, and hence, what action to take. |

### Description

A structure of this type is available in the audio renderer's progress function. It can be used by an application to monitor the status of the AV sync algorithm, or to implement an alternative sync algorithm.

## Audio Renderer API Functions

This section presents the Audio Renderer API application library functions.

| Name | Page |
|---|---|
| tmalArendAOGetCapabilities | 136 |
| tmolArendAOGetCapabilities | 136 |
| tmalArendAOGetCapabilitiesM | 137 |
| tmolArendAOGetCapabilitiesM | 137 |
| tmalArendAOGetNumberOfUnits | 138 |
| tmolArendAOGetNumberOfUnits | 138 |
| tmalArendAOOpen | 139 |
| tmolArendAOOpen | 139 |
| tmalArendAOOpenM | 140 |
| tmolArendAOOpenM | 140 |
| tmalArendAOClose | 141 |
| tmolArendAOClose | 141 |
| tmalArendAOInstanceSetup | 142 |
| tmolArendAOInstanceSetup | 142 |
| tmalArendAOStart | 143 |
| tmolArendAOStart | 143 |
| tmalArendAOStop | 144 |
| tmolArendAOStop | 144 |
| tmalArendAORenderBuffer | 145 |
| tmalArendAOInstanceConfig | 146 |
| tmolArendAOInstanceConfig | 146 |

## tmalArendAOGetCapabilities

```
tmLibdevErr_t tmalArendAOGetCapabilities(
   ptmalArendAOCapabilities_t   *pCap
);
```

## tmolArendAOGetCapabilities

```
tmLibdevErr_t tmolArendAOGetCapabilities(
   ptmolArendAOCapabilities_t    *pCap
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return a pointer to capabilities data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

This function fills in the pointer of a static structure, **ptmalArendAOCapabilities_t**, maintained by the renderer, to describe the capabilities and requirements of this library. This is achieved by calling **aoGetCapabilities** function.

The OL layer implements this by calling the AL layer and adding its overhead to the specifications of code size and data size found in the default capabilities structure. Implemented by a call to **tmalGetCapabilitiesM** with **unitName** set to the first unit (**unit0**).

## tmalArendAOGetCapabilitiesM

```
 tmLibappErr_t tmalArendAOGetCapabilitiesM (
   ptmalArendAOCapabilities_t  *pCap,
   unitSelect_t                 unitNumber
);
```

## tmolArendAOGetCapabilitiesM

```
 tmLibappErr_t tmolArendAOGetCapabilitiesM (
   ptmolArendAOCapabilities_t   *pCap,
   unitSelect_t                  unitNumber
);
```

### Parameters

| | |
|---|---|
| pCap | Pointer to pointer to an ArendAO capabilities structure at the appropriate level (AL or OL). |
| unitName | Select which unit. |

### Return Codes

| | |
|---|---|
| TMLIBDEV_OK | Success. |
| TMLIBDEV_ERR_NULL_PARAMETER | Can be asserted in debug mode. |
| TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW | Unit not supported in hardware. |

### Description

Used to find out about the AO hardware.

## tmalArendAOGetNumberOfUnits

```
tmLibappErr_t tmalArendAOGetNumberOfUnits (
   UInt32   *numberOfUnits
);
```

## tmolArendAOGetNumberOfUnits

```
tmLibappErr_t tmolArendAOGetNumberOfUnits (
   UInt32   *numberOfUnits
);
```

### Parameters

| | |
|---|---|
| `numberOfUnits` | Pointer to an integer describing the number of audio output units that are supported on the current hardware. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | In the debug version of the library, this assert is given if **numberOfUnits** is null. |

### Description

Finds the number of audio output units that are supported on the current hardware.

## tmalArendAOOpen

```
tmLibdevErr_t tmalArendAOOpen(
   Int    *instance
);
```

## tmolArendAOOpen

```
tmLibdevErr_t tmolArendAOOpen(
   Int    *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Returned if the renderer or audio hardware is already in use by someone else. |

Other error values may be returned from the underlying tmAO and BSP libraries.

### Description

Creates an instance of a renderer, and sets the instance variable given by aoOpen function. At the OL layer, memory is allocated for internal variables. Implemented by a call to **tm*x*lOpenM** with **unitName** set to the first unit (**unit0**).

## tmalArendAOOpenM

```
tmLibappErr_t tmalArendAOOpenM(
   Int           *instance,
   unitSelect_t   unitNumber
);
```

## tmolArendAOOpenM

```
tmLibappErr_t tmolArendAOOpenM(
   Int           *instance,
   unitSelect_t   unitNumber
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance. |
| `unitName` | Selects unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Unit not supported in hardware |
| `MLIBDEV_ERR_NO_MORE_INSTANCES` | Unit is already in use at the device library level. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Unit is in use at the AL or OL level. |
| `TMLIBDEV_ERR_MEMALLOC_FAILED` | Memory used for instance variable structure. |

Other errors might be returned if the allocation of the interrupt or hardware pins (on GPIO enabled devices) fail.

### Description

This function will open an instance of the selected AO device and assign the instance value. Using the tmAO device library, It opens an interrupt as appropriate for the specified unit with **intOpen** function.

## tmalArendAOClose

```
tmLibappErr_t tmalArendAOClose(
   Int    instance
);
```

## tmolArendAOClose

```
tmLibappErr_t tmolArendAOClose(
   Int    instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance returned by **tmalArendAOOpen** or **tmol-ArendAOOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Asserted if the renderer has not been opened by this instance. |

### Description

Shuts down this instance of the renderer by calling **aoClose**. At the OL layer, memory allocated for internal variables is freed. If InstanceSetup has been called, the interrupt service routine is active until this function is called.

## tmalArendAOInstanceSetup

```
tmLibappErr_t tmalArendAOInstanceSetup(
   Int                           instance,
   ptmalArendAOInstanceSetup_t   setup
);
```

## tmolArendAOInstanceSetup

```
tmLibappErr_t tmalArendAOInstanceSetup(
   Int                           instance,
   ptmalArendAOInstanceSetup_t   setup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance returned by **tmalArendAOOpen** or **tmol-ArendAOOpen**. |
| `setup` | Pointer to the setup structure. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| *(Errors from the device library)* | When a call into the device library fails, a unique error code is returned. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Returned if the renderer has not been opened by this instance. |
| `TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE` | Returned if the dataSubtype of the requested for-mat is unknown. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Returned if memory allocation for the silence buffer fails. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |

### Description

Initializes the renderer and sets up the audio out library with **aoInstanceSetup** using the appropriate format (specified in **setup**). Leaves renderer stopped. The OL layer calls the AL layer.

## tmalArendAOStart

```
tmLibappErr_t tmalArendAOStart(
   Int   instance
);
```

## tmolArendAOStart

```
tmLibappErr_t tmolArendAOStart(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance returned by **tmalArendAOOpen** or **tmol-ArendAOOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |

### Description

Starts the audio transmit for the specific instance. The interrupt service routine is instructed to begin processing audio data packets.

## tmalArendAOStop

```
tmLibappErr_t tmalArendAOStop(
   Int   instance
);
```

## tmolArendAOStop

```
tmLibappErr_t tmolArendAOStop(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| instance | Instance returned by **tmalArendAOOpen** or **tmolArendAOOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |

### Description

Stops the audio transmit for the specific instance. The interrupt is disabled and all outstanding packets are returned. The stop command may block for as long as the time it takes to play two audio packets. Stop will not return until the two packets currently installed in the hardware are no longer in use.

## tmalArendAORenderBuffer

```
tmLibappErr_t tmalArendAORenderBuffer(
   Int                instance,
   ptmAudioPacket_t   packet
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value as returned by **tmalArendAOOpen**. |
| `packet` | Pointer to a packet of audio data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_STREAM_MODE_CONFUSION` | This function is not to be used in streaming mode. Calling it when the **datain** function has been specified (to indicate streaming mode) returns this error code. |
| `AR_ERR_INSTANCE_BUSY` | Instance is not ready for new data. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |

### Description

In non-streaming mode, this function is used to hand over a buffer of samples to play. This buffer will be played using the settings assigned using tmalArendAOInstanceSetup. The completion callback function will be called when this buffer has finished playing and is free. At the same time, the **buffersInUse** field in the packet will be set to zero. In non-streaming mode, the audio renderer keeps an internal queue of four packets. Invocation of this function when the queue is full of packets to be played will have no effect and will return **AR_ERR_INSTANCE_BUSY**. True double-buffered operation is supported in by this function. Timestamp based synchronization is not supported in this mode.

## tmalArendAOInstanceConfig

```
tmLibappErr_t tmalArendAOInstanceConfig(
   Int              instance,
   arConfigParam_t  p,
   void*            val
);
```

## tmolArendAOInstanceConfig

```
tmLibappErr_t tmolArendAOInstanceConfig(
   Int                 instance,
   UInt32              flags,
   ptsaControlArgs_t   args
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmalArendAOOpen** or **tmolArendAOOpen**. |
| `p` | Address of parameter to be modified. |
| `val` | Pointer to the value to be used for modification. |
| `flags` | Not used. |
| `args` | Contains the parameter and value that are passed to the AL layer function |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `AR_ERR_UNSUPPORTED_COMMAND` | The parameter was not recognized. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |

### Description

These functions are used to configure parameters of the renderer while it is running. Unlike the configuration functions of task based modules, the audio renderer configuration functions act directly without using control queues. The supported parameters are are described as the **arConfigParam_t**, on page 127.

Chapter 7

# SPDIF Renderer (ArendSpdif) API

## SPDIF Audio Renderer API Overview

The SPDIF (Sony/Philips Digital Interface Format) audio renderer for TriMedia serves as a TSA-compatible interface between audio stream-producing modules and the outside world, using the SPDIF output that is now present on the TM-1300 and the TM-2700. The data format used for this interface is described in the international standard known as IEC60958. The IEC60958 standard describes two variants, one for consumer use and another for professional use. The term “SPDIF” applies to the consumer version of the standard, while the professional variant is called AES3, or AES/EBU. In this document, “SPDIF” is often used as a general term for these interfaces.

The SPDIF audio renderer is based on the standard AO audio renderer documented in Chapter 6, *Audio Renderer (ArendAO) API*. As a consequence, the renderer requires audio data to be packaged in **tmAvPacket_t** data structures. (For more information, see Chapter 4, *tmAvFormats.h: Multimedia Format Definitions*, in Book 3, *Software Architecture*, Part A).

The SPDIF audio renderer supports a streaming interface to the hardware, at the OL level. It can also be used as a function library to format a buffer for presentation to the hardware. This would happen at the AL level. To support the streaming interface, the SPDIF audio renderer installs an interrupt service routine and starts a task. The task reformats data from the TSA packet format it receives to the bitwise representation of the SPDIF format that is required by the SPDO hardware. Audio precision up to 24 bits is supported. Although the SPDIF hardware supports only the output of a stereo pair, the SPDIF renderer accepts multi-channel packets, just like the AO renderer. One channel pair is selected from these larger packets for presentation over the SPDIF interface. The SPDIF renderer offers full support for the so-called “C” bit, through an entry in the instance setup structure. It also allows users to control the “U” bit.

Like the AO renderer, the SPDIF renderer supports a sophisticated form of synchronization based on presentation times and a reference clock. The renderer is supplied as a library that can be used without restriction by owners of the TriMedia SDE.

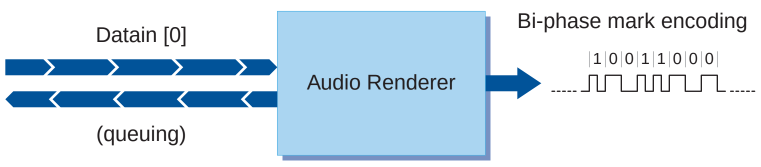


**Figure 3** Structure of the Audio Renderer

### Supporting SPDIF digital inputs using the audio digitizer:

With an appropriate board support package (BSP), the audio digitizer can support input from a digital audio source over an SPDIF (IEC60958) connection. The issues that must be addressed are demonstrated in the exolCopyAudio program, as well as in the DTV reference BSPs. The audio-in unit must be configured as a slave. A software PLL must be set up to lock the output rate to the input rate. And an interrupt must be provided to notify the application when the master SPDIF source changes. More information follows in the section discussing the progress function.

## Inputs and Outputs

The SPDIF Audio Renderer has one input and consumes buffers full of audio data. It returns the same buffers in an empty state. In the exchange, the data is presented at the SPDIF output. The SPDIF renderer calls the tmSPDO device library. The tmSPDO device library expects a BSP to be installed.

## Errors

Errors can be reported during the renderer's setup phase or at run time. Errors occurring during the setup phase are reported as non-zero return values from the API. In addition, the debugging version of the library uses assert mechanism to flag invalid inputs. Possible errors are listed in the descriptions of the API functions.

**Note**
We strongly advises that you bring up the SPDIF audio renderer using the _a (assert) version of the audio renderer and the device libraries. Many possible error conditions are flagged with assertions in these libraries.

For run-time errors, the audio renderer supports an error-reporting callback function. This function is called from the audio output interrupt service routine. None of the errors handled by the error callback function are fatal. The error function prototype is of the type **tsaErrorFunc_t**:

```
typedef tmLibappErr_t( *tsaErrorFunc_t )(
   Int instId,
   UInt32 flags,
   ptsaErrorArgs_t args
);
```

```
typedef struct tsaErrorArgs {
   Int       errorCode;
   Pointer   description;
} tsaErrorArgs_t, *ptsaErrorArgs_t;
```

You should provide handlers for these possible values of the errorCode:

### TMLIBAPP_ERR_UNDERRUN

The audio system requested data but nothing was available. This error could occur in several circumstances. The renderer's interrupt service routine always attempts to handle this case gracefully. More information about the source of the data is available in the description field. When an underrun error is logged, the renderer fills in the description field with a pointer to an array of three integers. The first member identifies the exact source of the error.

```
Values of errArg.description[0]:
0: The interrupt handler was locked out for too long.
1: The ISR could not receive data from the task.
Other error codes: Returned in the task when checking the input queue.
```

### TMLIBAPP_ERR_HIGHWAY_BANDWIDTH_ERR

Not enough bandwidth is available to service audio output on the TriMedia's internal data highway. This condition can possibly be remedied by reprogramming the bandwidth allocation MMIO register.

### ARENDSP_ERR_INVALID_BUFFER_SIZE

This message is given only by the task that formats data to be passed to the SPDO output. It occurs when an odd-sized buffer is passed to the output device. Usually, this case is handled correctly, but if it can't be, this error is logged and some data may be skipped.

### TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE

If the task formatting data for output receives a packet with a format that it does not understand, then the renderer reports this error and does not output the packet.

**Note**
Since an error function can be called from the context of an interrupt, it must not call printf or any other complex handler.

## Progress Function

When the renderer is operating, it can be configured to call a progress function. If you don't want to call a progress function, set the **progressFunc** member of the instance setup structure to Null. The **progressReportFlags** member of the default instance setup structure allows you to control when the progress function is called. Valid flags, defined in tmalArendSpdif.h, include these flags:

| | |
|---|---|
| `ARENDSP_PROGRESS_ReportCount` | Cause the progress function to be called in every ISR. |
| `ARENDSP_PROGRESS_ChangeSampleRate` | Cause the progress function to be called when the sample rate changes. |
| `ARENDSP_PROGRESS_SyncEventCorrect` | Cause the progress function to be called when the AV sync algorithm is active. |

The progress function has this prototype:

```
typedef tmLibappErr_t ( *tsaProgressFunc_t ) (
   Int                 instId,
   UInt32              flags,
   ptsaProgressArgs_t  args
);
```

where

```
typedef struct tsaProgressArgs {
   Int       progressCode;
   Pointer   description;
} tsaProgressArgs_t, *ptsaProgressArgs_t;
```

When called, the progress code will be some combination of the progress flags, as noted above. It is possible to call progress function with more than one flag set. When called with the progress flag **AREND_PROGRESS_ChangeSampleRate**, the description field contains a pointer to the new sample rate, as a **Float**. In the other cases, the description field of the **args** structure will usually contain a pointer to a structure of this type:

```
typedef struct tmArendSpdifControlInfo {
   UInt            numberOfSamples;
   ptmAvPacket_t   packet;
   Int             timeDiff;
   Int             muteTimer;
   arSyncState     syncState;
} tmArendSpdifControlInfo_t, *ptmArendSpdifControlInfo_t;
```

In the control info structure, the **numberOfSamples** field gives the number of samples that will be played when the buffer installed by this interrupt is played. Knowledge of this number allows you to implement a simple form of synchronization using the mechanism of the DDS clock synthesizers available on TriMedia. For example, the output clock can be phase-locked to an external source at the audio input. The packet member of this structure is the address of the audio packet that is being installed in this interrupt. The **timeDiff** member of the structure implements the more sophisticated type of synchronization appropriate for time-stamped data streams. The **syncState** and **muteTimer** members monitor the operation of the internal synchronization mechanism. They are described in the API reference entry for this structure.

## How to Use the Audio Renderer

The SPDIF audio renderer is to be used as a streaming driver accessed from the OL layer. The sample program **exolArendSpdif** demonstrates streaming operation at the OL layer, using the TSSA default functions and pSOS.

To use the audio renderer, follow these steps:

1. Call the Open function to create an instance.
2. Obtain a pointer to an instance setup structure and fill it with your data. The instance setup structure completely describes the operation of the renderer. Refer to tsa.h for the definition of the structure **ptsaDefaultInstanceSetup_t**.

In particular, you will fill in the **format** structure. Entries are provided for a number of callback functions. When working at the OL layer, the datain function is provided as a default.

3. Call the InstanceSetup function to register these settings with the renderer.
4. Call the Start function. This will cause the renderer to expect data and consequently, to log ensuing errors if data is not present.
5. To change the mode of operation while running, call InstanceConfig.
6. When you are done, call Stop and then Close.

## How the Audio Renderer Works

The audio renderer installs an interrupt service routine which manages a small queue of buffers formatted especially for the SPDO unit. The size of these buffers is determined by the **bufferSize** member in the instance setup structure. This size is given in samples. It must be a multiple of 192 (the SPDIF frame size). The actual amount of memory allocated (in bytes) is 32 × **bufferSize**. The interrupt service routine does very little besides servicing the hardware.

The renderer also spawns a task that receives packets in TSA format, reformats them for the SPDO output hardware, and passes them along to the ISR.

A more complete description of the use of the audio renderer in streaming mode is found in the example program exolArendSpdif. As with any OL layer component conforming to the streaming architecture, the setup happens through the structures passed to **tmolArendSpdifInstanceSetup**. Communication queues are allocated and buffers are placed in the empty queue. As the source component is initialized with the same pair of queues, data exchange begins as soon as **tmolArendSpdifStart** is called.

## Formats in the Audio Renderer

The formats supported by the SPDIF audio renderer are determined by a call to **tmAOGetCapabilities**, which queries the board support package. The SPDIF audio renderer currently supports 2-, 6-, and 8-channel PCM formats in 16- and 32-bit variants.

The OL version of the audio renderer can change its format on-the-fly. It is possible (and acceptable) to specify no format at instance setup, instead relying on the format being passed in the first data packet. The audio renderer will change its format based on the format specified in any packet. Because the format change function cannot be called from the interrupt service routine, it is called from the sending component's dataout function. This may result in a few queued packets being played with the wrong format.

## Synchronization Overview

The SPDIF renderer operates in exact analogy to the AO audio renderer.

The SPDIF renderer includes a number of services to be used for synchronization. It also includes specific code to synchronize audio and video streams, based on a reference clock and time-stamped packets. The AV sync code is used, for example, in the TriMedia DTV reference application. Other forms of sync, such as “broadcast sync” and “AA sync” are supported with the application’s ability to change the audio sample rate and the reports given by the renderer in the progress function. The interface includes a **syncMode** parameter. When this is set to **AR_Sync_None**, sync is disabled and all packets are played as received. In **AR_Sync_trigger** mode, the renderer expects the first packet it sees to be time-stamped, and this packet is held until the reference clock is greater than the time stamp. The **AR_Sync_skip** mode is the most powerful.

## AV Sync Details

There are many details involved in the renderer’s AV synchronization scheme. When the **AR_Sync_trigger** mode is used, the operation is almost trivial. An example of the complete **AR_Sync_skip** mechanism is demonstrated in the DTV reference app, ATSCbasic. The mechanism is still conceptually simple, as described below:

In order for the AV sync mechanism to be enabled, the user must explicitly enable the mechanism by setting the syncMode to **AR_Sync_skip**. Also, a reference clock must be installed at instance setup, and packets to be synchronized must contain a valid timestamp. With these pre-requisites met, the algorithm attempts to present the packet at the correct time. This behavior is controlled by the user’s specification of the **syncMode** and the **timeThreshold**. In very general terms:

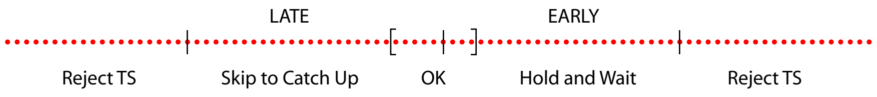
1. If the time stamp matches the clock to the accuracy of the **timeThreshold** field, the packet will be played. The user is informed of this condition through the progress function, and the **syncState** value of **AR_SyncState_CorrectionAppropriate** informs him that this is a good time to fine tune the audio sample clock, for example, with the DDS.
2. If the time stamp is earlier than the current clock value, the packet is returned without being fully played.
3. If the time stamp is in the future, the packet is held until the clock reaches the value of the timestamp.
4. If the difference between the time stamp and the clock is greater than 16 times the **timeThreshold**, then the time stamp data is assumed to be erroneous and it is ignored. Note that it is this parameter that determines the number of packets that must be available in the system. If the audio renderer can hold a packet for 16 times the threshold, the rest of the system must be prepared for this possibility.

### timeDiff

The fundamental measure of AV sync is the comparison between the Presentation Time Stamp (PTS) and the clock, sometimes called the Program Clock Reference (PCR). This comparison is made using the difference (**timeDiff**) between these two clocks, each represented by a 32-bit number. When the difference is positive, the packets are ahead of the clock. Since humans prefer late audio to early audio, the skipping algorithm is biased to return slightly negative time differences.

## timeThreshold

The **timeThreshold** member of the instance setup structure sets how far out of sync the clock and the PTS can be before drastic action is taken. When the difference (**timeDiff**) is less than the threshold, the audio can track the video without gaps in the audio data. In MPEG applications, the PCR usually runs at 90KHz, and a threshold of 3000 works well.



When a timestamp is encountered that is inside of the rejection window, but outside of the acceptable window, and early, it is held and silence is played until the timestamp matches the reference clock. If it is late, a short packet of silence is played (64 samples) and, the next packet is retrieved.

As a further conservative measure, timestamps must indicate the necessity of skipping or waiting three times consecutively before any action is taken.

As mentioned above, the human bias against early audio causes the threshold to be asymmetric around zero. A value of timeThreshold/2 is used for positive time differences.

## Rejecting Bad Time Stamps

When the absolute value of the time difference is greater than sixteen times the threshold, the software assumes the clock or the time stamp is bad, and the packet is played as if it were not time stamped.

## Holding a Packet When Ahead

When the time difference is greater than the half threshold (but less than 16 times the threshold), the packet is ahead of the clock and the renderer will hold this packet until the clock catches up. The progress function is called with the **syncState** set to **AR_SyncState_Waiting**. When a packet is being held, the audio renderer output is muted, the mute counter is initialized to the value of **muteCounterInit** as specified in the instance setup structure, and silence is played. Note that this mechanism places a minimum on the number of packets that are available in a system. If the threshold is set high, and too few packets are circulated in the queues, it may be possible for all of the packets to end up held by the audio system because of this case. The threshold should be set appropriately and enough packets should be available so that this cannot become an issue.

### Playing Short to Catch Up

When the time difference is less than the negative threshold (but not less than 16 times the negative threshold), the packet is behind the clock, and the audio needs to catch up. The renderer catches up by playing only the minumum number of samples from the packet. This number is 64 stereo 16-bit samples, but it is smaller for multi-word samples. Note that it might be possible to break this mechanism if you routinely use very short buffers with the renderer. The progress function is called with the **syncState** set to **AR_SyncState_Skipping**. As in the "holding" case, the audio renderer output is muted, the mute counter is initialized to the value of **muteCounterInit** as specified in the instance setup structure, and silence is played while the renderer is skipping.

### Adapting the Sample Rate

When the time difference is within the bounds set by the threshold, the application is given the opportunity to drive the remaining difference to zero using some sort of a linear controller in a feedback loop. The controller can be very simple: If the output clock is based on the TriMedia DDS, the output sample rate linearly follows the DDS control word. The DDS control word can be modified according to an equation like this:

$$\text{newDDS} = (1 - \text{Kp} \times \text{timeDiff}) \times \text{originalDDS}.$$

This sort of an update is normally done when the progress function is called with **syncState** equal to **AR_SyncState_CorrectionAppropriate**. For more information, see the example code in the ATSCbasic application.

### Sync Delay

It is often useful to add an offset to the PCR (clock) when computing the time difference. The audio renderer can accept this offset as the **syncDelay**. It can be specified at instance setup, or changed on the fly using the **AR_SET_DELAY** command to the **InstanceConfig** function. The **syncDelay** is added to the **timeDiff**, so a positive delay will move the packet forward in time.

As an example, the **syncDelay** is useful when the video renderer is also locking to the same PCR. The video renderer can only lock to an accuracy of one frame, but it can measure an offset to the clock of less than one frame. In the DTV applications, the video renderer passes this offset to the audio renderer as a the syncDelay, and the audio renderer uses this to compute the time difference.

But when the delay tends to jump abruptly, it might be appropriate to filter the delay so that artifacts in the sound are less noticeable. To do this, do not use the audio renderer's delay parameter. Instead, add the filtered delay value to the time difference reported in the progress function. A filter like this will be updated only when a packet with a timestamp is processed.

### Muting

Whenever the renderer is muted, a counter internal to the renderer is initialized to the value specified as **muteCounterInit** at instance setup. At the moment mute is enabled, the counter begins to count down. Mute is disabled when the counter gets back to zero. If you do not want this feature, simply set **muteCounterInit** to zero. But the mute counter is also used when muting is enabled by the AV sync mechanism. The counter is used to avoid the situation where several short periods of mute are heard while the AV sync algorithm locks up. It is better to stay muted until everything is stable.

## SPDIF Audio Renderer API Data Structures

This section presents the SPDIF Audio Renderer data structures.

| Name | Page |
|---|---|
| arendSpdifConfigParam_t | 157 |
| arendSpdifProgressFlags_t | 160 |
| arendSpdifSyncMode_t | 161 |
| arendSpdifSyncState_t | 161 |
| tmalArendSpdifChannelStatus_t | 162 |
| tmalArendSpdifCapabilities_t | 163 |
| tmolArendSpdifCapabilities_t | 163 |
| tmalArendSpdifInstanceSetup_t | 164 |
| tmolArendSpdifInstanceSetup_t | 164 |
| tmArendSpdifControlInfo_t | 166 |

## arendSpdifConfigParam_t

```
typedef enum {
   ARENDSP_MUTE,
   ARENDSP_SET_MUTE,
   ARENDSP_GET_MUTE,
   ARENDSP_SET_DELAY,
   ARENDSP_SET_SAMPLE_RATE,
   ARENDSP_GET_SAMPLE_RATE,
   ARENDSP_SET_SYNC_MODE,
   ARENDSP_GET_SYNC_MODE,
   ARENDSP_SET_BAD_TIMESTAMP_THRESHOLD,
   ARENDSP_SET_SYNC_THRESHOLD,
   ARENDSP_SPDIF_SET_DATA_MODE,
   ARENDSP_SPDIF_PCM_MODE,
   ARENDSP_SPDIF_SET_COPYRIGHT_INFO,
   ARENDSP_SPDIF_SET_COPYRIGHT_BIT,
   ARENDSP_SPDIF_UNSET_COPYRIGHT_BIT,
   ARENDSP_SET_CBITS,
   ARENDSP_SET_UBITS,
   ARENDSP_SET_CHANNEL_PAIR,
   ARENDSP_GET_CHANNEL_PAIR,
} arendSpdifConfigParam_t;
```

### Description

Enumerates the commands used by the InstanceConfig function. When you call **tmo-lArendSPDIFInstanceConfig**, you pass a configuration command which includes one of these command values and a pointer to a parameter. If the command is a 'set' command, you pass the appropriate parameter value also. If the command is a 'get' command, the parameter value is returned in the parameter.

The following list tells you the parameter types for a call to **tmolArendSPDIFInstance-Config**.

### Fields

| | |
|---|---|
| `ARENDSP_MUTE` | The **MUTE** command toggles the mute state. |
| `ARENDSP_SET_MUTE` | When you use **SET_MUTE**, pass the Boolean value directly as the parameter. |
| `ARENDSP_GET_MUTE` | When you use **GET_MUTE**, pass the address of a Boolean variable. |
| `ARENDSP_SET_DELAY` | Specify delay in units of the currently installed TSA clock and pass it directly as the parameter. |
| `ARENDSP_SET_SAMPLE_RATE` | Pass the address of the specified rate (float) as the parameter. |

| | |
|---|---|
| `ARENDSP_GET_SAMPLE_RATE` | Pass the address to receive the rate (float) as the parameter. |
| `ARENDSP_SET_SYNC_MODE` | Pass the address of the mode as the parameter. |
| `ARENDSP_GET_SYNC_MODE` | Pass the address of the mode as the parameter. |
| `ARENDSP_SET_BAD_TIMESTAMP_THRESHOLD` | When sync skip mode is active, packets can be rejected if their timestamp is too far from the current clock value. This configuration command allows the threshold of rejection to be changed dynamically. The default value is 48,000 ticks. Pass the address of the threshold as the parameter. |
| `ARENDSP_SET_SYNC_THRESHOLD` | This configuration command dynamically adjusts the difference between the reference clock and a timestamp on a packet that is considered “close enough” for playback. Pass the address of the threshold as the parameter. |
| `ARENDSP_SPDIF_SET_DATA_MODE` | This configuration command forces the SPDIF C bit to identify the channel as non-PCM data. The command takes no parameter. |
| `ARENDSP_SPDIF_PCM_MODE` | This configuration command (the inverse of the above command) forces the SPDIF C bit to identify the channel as PCM data. The command takes no parameter. |
| `ARENDSP_SPDIF_SET_COPYRIGHT_INFO` | When the renderer is operating in consumer mode, the category code and the L bit can be set using this command. The parameter of the control argument points to an integer that will be dereferenced and cast to the type **UInt8**. The low seven bits are the category code (as defined by IEC60958, and using the constants defined in tmalArendSpdif.h). The eighth bit (mask 0x80) is used to set the “L” bit. |
| `ARENDSP_SPDIF_SET_COPYRIGHT_BIT` | When the renderer is operating in consumer mode, this command sets the copyright bit. The command takes no parameter. |
| `ARENDSP_SPDIF_UNSET_COPYRIGHT_BIT` | When the renderer is operating in consumer mode, this command clears the copyright bit. The command takes no parameter. |
| `ARENDSP_SET_CBITS` | This command sets any field of the officially-defined C bit structures. Here, the command parameter is treated as an array of pointers, the first pointing to a “field” flag that allows you to control the A and B fields of the C bit separately, |

| | |
|---|---|
| | or as a unit, and the second being a pointer to a C bit union structure (**tmalArendSpdifChannel-Status_t**). |
| `ARENDSP_SET_UBITS` | This command sets any field of the U bit structure. Here, the command parameter is treated as an array of pointers, the first pointing to a “field” flag that allows a user to control the A and B fields of the U bit separately, or as a unit, and the second being a pointer to an array of 24 **UInt8**s (192 bits). |
| `ARENDSP_SET_CHANNEL_PAIR` | When the renderer has more than two channels of input, it can select one channel pair from the input using this command. Pass the command parameter as a pointer to an integer channel pair specifier. |
| `ARENDSP_GET_CHANNEL_PAIR` | When the renderer has more than two channels of input, it can select one channel pair from the input using this command. The command parameter is returned as a pointer to an integer channel pair specifier. |

## arendSpdifProgressFlags_t

```
typedef enum {
   ARENDSP_PROGRESS_ReportCount,
   ARENDSP_PROGRESS_ChangeSampleRate,
   ARENDSP_PROGRESS_SyncEventCorrect,
   ARENDSP_PROGRESS_EndOfStream,
   ARENDSP_PROGRESS_ChangeFormat
} arendSpdifProgressFlags_t;
```

### Fields

| | |
|---|---|
| `ARENDSP_PROGRESS_ReportCount` | Causes the progress function to be called at every interrupt to report the count of samples played, using the **tmArendSpdifControlInfo_t** structure. |
| `ARENDSP_PROGRESS_ChangeSampleRate` | Causes the progress function to be called if the sample rate changes, allowing the application's sync algorithm to be informed of sample rate changes. Note that the progress function is not called for the initial installation of the sample rate, because the sample rate is set before the progress function variable is set. |
| `SyncEventCorrect` | Causes the progress function to be called if a timestamp was detected in a received packet, and hence a sync event was triggered. This progress event is critical to the implementation of AV sync algorithms. When called, the progress parameter is a pointer to a **tmArendSpdifControlInfo_t** structure. |

### Description

The renderer can call the progress function under a number of conditions. **EndOfStream** and **ChangeFormat** are TSSA standard progress occasions.

## arendSpdifSyncMode_t

```
typedef enum {
   ARENDSP_Sync_None,
   ARENDSP_Sync_trigger,
   ARENDSP_Sync_skip,
} arendSpdifSyncMode_t;
```

### Description

Synchronization processing can be disabled (**AR_Sync_None**), or it can be enabled in one of two modes. In “trigger” mode, the renderer expects the first packet it receives to be time-stamped, and this packet is held until the reference clock matches the timestamp. After that, sync information is ignored, and all packets are played. The trigger is reset by stopping and starting the renderer.

In “sync skip” mode, the renderer uses the algorithm described under AV Sync Details. Packets are constantly checked for their relation to the reference clock, and they are always presented within the window specified.

## arendSpdifSyncState_t

```
typedef enum {
   ARENDSP_SyncState_NoAction,
   ARENDSP_SyncState_Waiting,
   ARENDSP_SyncState_Skipping,
   ARENDSP_SyncState_CorrectionAppropriate,
} arSyncState_t;
```

### Description

When synchronization processing is enabled, the ‘sync state’ is communicated to the application using the **tmArendSpdifControlInfo_t** structure as passed in the progress function. The skipping and waiting states tell the application that the timestamp of the packet to be played is outside of the legal window.

When your application sees **ARENDSP_SyncState_CorrectionAppropriate** state, the packet is within its window and further action is up to your application. It is “appropriate” to “correct” the sample rate clock.

## tmalArendSpdifChannelStatus_t

```
typedef union arendSpdif_channel_status_t{
  struct{                           // consumer:
    Bool                               consumerMode;
    Bool                               PCM;
    Bool                               copyright;
    arendSpdifConsumerFormat_t         format;
    Bool                               LcopyBit;
    UInt8                              CategoryCode;
    UInt8                              SourceNumber;
    UInt8                              ChannelNumber;
    arendSpdifConsumerSampleRate_t     SamplingFrequency;
    arendSpdifConsumerClockAccuracy_t  ClockAccuracy;
    arendSpdifConsumerWordLength_t     WordLength;
  } consumer;
  struct {                              // Professional
    Bool                                   consumerMode;
    Bool                                   PCM;
    arendSpdifProfessionalEmphasis_t       emphasis;
    Bool                                   SRateUnLocked;
    arendSpdifProfessionalSampleRate_t     SamplingFrequency;
    arendSpdifProfessionalChannelMode_t    ChannelMode;
    arendSpdifProfessionalCUserBitsMgt_t   UserBitsMgt;
    arendSpdifProfessionalCUseOfAuxBits_t  UseOfAuxBits;
    UInt8     SourceWordLengthAndHistory;
    Bool      DigitalAudioReferenceSignal;
    char      Origin1;
    char      Origin2;
    char      Origin3;
    char      Origin4;
    char      Destination1;
    char      Destination2;
    char      Destination3;
    char      Destination4;
    Int32     LocalSampleAddressCode;
    Int32     TimeOfDayCode;
    char      ReliabilityFlags;
    char      CRC;
  } professional;
} tmalArendSpdifChannelStatus_t, *ptmalArendSpdifChannelStatus_t;
```

### Description

The C Bits of an IEC60958-compliant data stream can have different meanings depending upon whether they are used in professional (AES3, or AES/EBU) or consumer (SPDIF) modes. This union reflects this fact. The specific types referenced in the union are defined as enumerated types in tmalArendSpdif.h to provide strong type checking.

## tmalArendSpdifCapabilities_t

```
typedef struct tmalArendSpdifCapabilities_t {
   ptsaDefaultCapabilities_t   defaultCapabilities;
   Int32                       max_srate;
   Int32                       min_srate;
   UInt32                      audioAdapters;
   Int32                       granularityOfAddress;
   Int32                       granularityOfSize;
   Int32                       minBufferSize;
   UInt32                      mmioBaseAddress;
} tmalArendSpdifCapabilities_t; *ptmalArendSpdifCapabilities_t;
```

## tmolArendSpdifCapabilities_t

```
typedef tmalArendSpdifCapabilities_t
tmolArendSpdifCapabilities_t, *ptmolArendSpdifCapabilities_t;
```

### Fields

| | |
|---|---|
| `defaultCapabilities` | In compliance with the application library architecture, this is a pointer to a standard capabilities structure. |
| `max_srate` | Minimum sample rate [Hz]. |
| `min_srate` | Maximum sample rate [Hz]. |
| `audioAdapters` | Will be **aaaDigitalOutput**, unless extra BSP support is implemented. |
| `granularityOfAddress` | Number of LSBs that should be zero (for example, 6 bits for 64-byte alignment). |
| `granularityOfSize` | Number of LSBs that should be zero in the size field (size is the number of samples). |
| `minBufferSize` | Minimum buffer size (samples). |
| `mmioBaseAddress` | Set from the BSP and used to determine the base address of the underlying hardware. |

### Description

This type (either version, above) holds a list of capabilities. The audio renderer maintains a structure of this type to describe itself. A user can retrieve the structure's address by calling **tmalArendSpdifGetCapabilities** or **tmolArendSpdifGetCapabilities**. Notice that the AL and the OL layer structures are identical, except for the extensions to the default capabilities structure made in the OL layer (tsa.h).

## tmalArendSpdifInstanceSetup_t

```
typedef struct tmalArendSpdifInstanceSetup_t {
   ptsaDefaultInstanceSetup_t      defaultSetup;
   Int32                           bufferSize;
   tmAudioAnalogAdapter_t          output;
   Int                             muteCounterInit;
   UInt32                          syncThreshold;
   Int                             syncDelay;
   arSyncMode_t                    syncMode;
   Int32                           badTimestampThreshold;
   ptmalArendSpdifChannelStatus_t  channelStatusA;
   UInt8                          *userBitsA;
   ptmalArendSpdifChannelStatus_t  channelStatusB;
   UInt8                          *userBitsB;
} tmalArendSpdifInstanceSetup_t;
```

## tmolArendSpdifInstanceSetup_t

```
typedef tmalArendSpdifInstanceSetup_t
tmolArendSpdifInstanceSetup_t, *ptmolArendSpdifInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `defaultSetup` | Refer to tsa.h for more information. The function pointers (error func, datain func) are taken from here, as is the format. |
| `bufferSize` | Specifies the size of the buffers used internally to communicate between the task and the ISR. This value must be a multiple of 1536 bytes (192×8). |
| `output` | Must be **aaaDigitalOutput**, unless extra BSP support is implemented and indicated by capabilites structure. |
| `muteCounterInit` | Whenever the audio renderer is muted, either by user command or by loss of AV sync, a counter internal to the renderer is initialized to this value. It is then decremented when the mute condition is lifted. Only when the counter goes to zero is the mute actually ended. Set to zero to disable this feature. |
| `syncThreshold` | When the difference between the current time and the timestamp on a packet exceeds this threshold, packets are held or skipped to correct the loss of sync. This mechanism is activated if |

| | |
|---|---|
| | (1) A clock is installed at instance setup or (2) Valid timestamps are provided on received packets. A value equal to 30ms is usually appropriate. |
| `syncDelay` | An offset, given in units of the installed TSA clock that is used to compute audio video (AV) sync. Positive values delay the playback of packets. See page 155. |
| `syncMode` | See **arendSpdifSyncMode_t** on page 161. |
| `badTimestampThreshold` | When sync skip mode is active, packets can be rejected if their timestamp is too far from the current clock value. This configuration command allows the threshold of rejection to be changed dynamically. Default value is 48,000 ticks. |
| `channelStatusA` | |
| `channelStatusB` | These structures configure the “C bits” as described in the IEC60958 standard. The union definition allows convenient access to each field. Since the A and B frames can be different, two pointers are provided. |
| `userBitsA`, `userBitsB` | These pointers are used to configure the “U bits” as described in the IEC60958 standard. Since the use of the U bits is left to the user, a pointer to an array of 192 bits (24 bytes) is used. Since the A and B frames can be different, two pointers are provided. |

## Description

A structure of this type is passed to **tmalArendSpdifInstanceSetup** or to **tmolArendSpdifInstanceSetup**.Using the standard tmal (TriMedia Application Library) model, you can configure the renderer at the AL or OL layer. The OL layer simply calls the AL layer.

## tmArendSpdifControlInfo_t

```
typedef struct tmArendSpdifControlInfo {
   UInt            numberOfSamples;
   ptmAvPacket_t   packet;
   Int             timeDiff;
   Int             muteTimer;
   arSyncState_t   syncState;
} tmArendSpdifControlInfo_t, *ptmArendSpdifControlInfo_t;
```

### Fields

| | |
|---|---|
| `numberOfSamples` | The number of samples that will be played in this invocation of the audio renderer ISR. |
| `packet` | Pointer to the packet that will be played in this invocation of the audio renderer ISR. |
| `timeDiff` | Difference in time, give in the units of the currently installed TSA clock, between the time stamp of the current packet, and its expected presentation time, with the addition of the syncDelay. See page 153. |
| `muteTimer` | The current value of the mute timer. |
| `syncState` | Tells the progress function what decision the renderer has made about sync, and hence, what action to take. |

### Description

A structure of this type is available in the audio renderer's progress function. It can monitor the status of the AV sync algorithm or implement an alternative sync algorithm.

## SPDIF Audio Renderer API Functions

This section presents the Audio Renderer API application library functions.

## tmalArendSpdifGetCapabilities

```
tmLibdevErr_t tmalArendSpdifGetCapabilities(
   ptmalArendSpdifCapabilities_t   *pCap
);
```

## tmolArendSpdifGetCapabilities

```
tmLibdevErr_t tmolArendSpdifGetCapabilities(
   ptmolArendSpdifCapabilities_t   *pCap
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return a pointer to capabilities data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

The function fills the pointer of a static structure, **ptmalArendSpdifCapabilities_t**, maintained by the renderer, to describe the capabilities and requirements of this library. It calls **aoGetCapabilities** function.

The OL layer implements this by calling the AL layer and adding its overhead to the specifications of code size and data size found in the default capabilities structure. The function is implemented by a call to **tmalGetCapabilitiesM** with **unitName** set to the first unit (**unit0**).

## tmalArendSpdifGetCapabilitiesM

```
tmLibappErr_t tmalArendSpdifGetCapabilitiesM (
   ptmalArendSpdifCapabilities_t  *pCap,
   unitSelect_t                    unitNumber
);
```

## tmolArendSpdifGetCapabilitiesM

```
tmLibappErr_t tmolArendSpdifGetCapabilitiesM (
   ptmolArendSpdifCapabilities_t   *pCap,
   unitSelect_t                     unitNumber
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return capabilities data at the appropriate level (AL or OL). |
| `unitName` | Select which unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Unit not supported in hardware. |

### Description

Finds out about the SPDIF hardware.

## tmalArendSpdifGetNumberOfUnits

```
tmLibappErr_t tmalArendSpdifGetNumberOfUnits (
   UInt32  *numberOfUnits
);
```

## tmolArendSpdifGetNumberOfUnits

```
tmLibappErr_t tmolArendSpdifGetNumberOfUnits (
   UInt32  *numberOfUnits
);
```

### Parameters

| | |
|---|---|
| numberOfUnits | Pointer to a variable in which to return the number of audio output units that are supported on the current hardware. |

### Return Codes

| | |
|---|---|
| TMLIBDEV_OK | Success. |
| TMLIBDEV_ERR_NULL_PARAMETER | In the debug version of the library, this asserts if **numberOfUnits** is null. |

### Description

Find the number of audio output units that are supported on the current hardware.

## tmalArendSpdifOpen

```
tmLibdevErr_t tmalArendSpdifOpen(
   Int   *instance
);
```

## tmolArendSpdifOpen

```
tmLibdevErr_t tmolArendSpdifOpen(
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | The renderer or audio hardware is already in use. |

Other error values may be returned from the underlying tmSPDO and BSP libraries.

### Description

Creates an instance of a renderer, and sets the instance variable given by **aoOpen** function. At the OL layer, memory is allocated for internal variables. The function is implemented by a call to **tmxlOpenM** with **unitName** set to the first unit (**unit0**).

## tmalArendSpdifOpenM

```
tmLibappErr_t tmalArendSpdifOpenM(
   Int           *instance,
   unitSelect_t   unitNumber
);
```

## tmolArendSpdifOpenM

```
tmLibappErr_t tmolArendSpdifOpenM(
   Int           *instance,
   unitSelect_t   unitNumber
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer to instance variable, stored as an integer. This variable, assigned in open, is used to access the unit in subsequent calls. |
| `unitName` | Select which unit. |

### Return Codes

| | |
|---|---|
| `TMLIBDEV_OK` | Success. |
| `TMLIBDEV_ERR_NULL_PARAMETER` | Can be asserted in debug mode. |
| `TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW` | Unit not supported in hardware. |
| `MLIBDEV_ERR_NO_MORE_INSTANCES` | Unit is already in use at the device library level. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Unit is in use at the AL or OL level. |
| `TMLIBDEV_ERR_MEMALLOC_FAILED` | Memory used for instance variable structure. |

Other errors might be returned if the allocation of the interrupt or hardware pins (on GPIO enabled devices) fail.

### Description

This function will open an instance of the selected SPDO device and assign the instance value. Using the tmSPDO device library, It opens an interrupt as appropriate for the specified unit with **intOpen** function.

## tmalArendSpdifClose

```
tmLibappErr_t tmalArendSpdifClose(
   Int   instance
);
```

## tmolArendSpdifClose

```
tmLibappErr_t tmolArendSpdifClose(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmalArendSpdifOpen** or **tmolArendSpdifOpen.** |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Asserted if the renderer has not been opened by this instance. |

### Description

Shuts down this instance of the renderer by calling **aoClose**. At the OL layer, memory allocated for internal variables is freed. If InstanceSetup has been called, the interrupt service routine is active until this function is called.

## tmalArendSpdifInstanceSetup

```
tmLibappErr_t tmalArendSpdifInstanceSetup(
   Int                                instance,
   ptmalArendSpdifInstanceSetup_t     setup
);
```

## tmolArendSpdifInstanceSetup

```
tmLibappErr_t tmalArendSpdifInstanceSetup(
   Int                                instance,
   ptmalArendSpdifInstanceSetup_t     setup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmalArendSpdifOpen** or **tmolArendSpdifOpen**. |
| `setup` | Pointer to the setup structure. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | The renderer has not been opened by this instance. |
| `TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE` | The data subtype of the requested format is unknown. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Memory allocation for the silence buffer fails. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |
| *Errors from the device library* | When a call into the device library fails, a unique error code is returned. |

### Description

Initializes the renderer and sets up the audio out library with **aoInstanceSetup** using the appropriate format (specified in s). Leaves renderer stopped. The OL layer calls the AL layer.

## tmalArendSpdifStart

```
tmLibappErr_t tmalArendSpdifStart(
   Int   instance
);
```

## tmolArendSpdifStart

```
tmLibappErr_t tmolArendSpdifStart(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmalArendSpdifOpen** or **tmolArendSpdifOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |

### Description

Starts the audio transmit for the specific instance. The interrupt service routine is instructed to begin processing audio data packets.

## tmalArendSpdifStop

```
tmLibappErr_t tmalArendSpdifStop(
   Int   instance
);
```

## tmolArendSpdifStop

```
tmLibappErr_t tmolArendSpdifStop(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmalArendSpdifOpen** or **tmolArendSpdifOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |

### Description

Stops the audio transmit for the specific instance. The interrupt is disabled and all outstanding packets are returned. The stop command may block for as long as the time it takes to play two audio packets. Stop will not return until the two packets currently installed in the hardware are no longer in use.

## tmalArendSpdifInstanceConfig

```
tmLibappErr_t tmalArendSpdifInstanceConfig(
   Int                 instance,
   ptsaControlArgs_t   args
);
```

## tmolArendSpdifInstanceConfig

```
tmLibappErr_t tmolArendSpdifInstanceConfig(
   Int                 instance,
   UInt32              flags,
   ptsaControlArgs_t   args
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmalArendSpdifOpen** or **tmolArendSpdifOpen**. |
| `flags` | Not used. |
| `args` | Contains the parameter and value that are passed to the AL layer function. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `AR_ERR_UNSUPPORTED_COMMAND` | The parameter was not recognized. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can assert in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Can assert in debug mode. |

### Description

These functions are used to configure parameters of the renderer while it is running. Unlike the configuration functions of task-based modules, the audio renderer configuration functions act directly without using control queues. The supported parameters are are described as the **arMode_t** on page 126.

## tmalArendSpdifFormatTemplate

```
tmLibappErr_t tmalArendSpdifFormatTemplate(
   Int      instance,
   Int32   *templateBuffer
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmalArendSpdifOpen** or **tmolArendSpdifOpen**. |
| `templateBuffer` | Points to a buffer that will be used by the SPDO hardware to transmit a valid SPDIF frame. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Fills in the C bits and U bits of a buffer for transmission via the SPDO hardware. The desired configuration of the C and U bits are read from the instance variable. The templateBuffer must point to memory enough for at least one block (1536 bytes). The size of this memory is specified by the bufferSize parameter passed at instanceSetup.

A default template (including preambles) is copied to the template. Then the two C bit structures and the two U bit structures are expanded into the template.

## tmalArendSpdifFormatBuffer

```
tmLibappErr_t tmalArendSpdifFormatBuffer (
   Int              instance,
   ptmAvPacket_t    inPacket,
   Int32           *outPointer,
   Int32            bytesAvailableInOutputBuffer,
   Int32           *bytesCopiedToOutput,
   Int32           *bytesUsedFromInput
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmalArendSpdifOpen** or **tmolArendSpdifOpen**. |
| `inPacket` | A TSSA audio packet containing input data. |
| `outPointer` | Points to a buffer that will be used by the SPDO hardware to transmit a valid SPDIF frame. |
| `bytesAvailableInOutputBuffer` | Used to avoid writing past the end of the output buffer. |
| `bytesCopiedToOutput` | Pointer to an integer to be updated with the number of bytes that have been copied into the output buffer. This number will be 8 times the number of stereo samples played, as one SPDIF output frame is 8 byte long. Allows the caller to update the output pointer. |
| `bytesUsedFromInput` | Pointer to an integer that will be updated with the number of bytes that have been read from the input buffer. This number will take into account the dependency of the size of an input frame on the number of channels and on the size of the data (16 or 32 bits). Allows the caller to update the input pointer. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Returned if the function completes successfully. |
| `TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE` | Returned if the input packet's data subtype is not one supported (as listed in capabilities structure). |

### Description

Used to transfer audio data from an input TSA packet to an output SPDO format packet. Supports partial filling of the output buffer, but must consume the input buffer completely. If the input buffer does not fit into the output buffer, then the contents of the

input buffer will be dropped, and the error callback function will be called to signal the error **ARENDSP_ERR_INVALID_BUFFER_SIZE**.
The output buffer is assumed to have been previously setup using **tmalArendSpdifFormatTemplate**. This function does not touch the C bits or U bits. Data is OR'd into the buffer.

# Chapter 8

# Simple Audio Mixer (AmixSimple) API

## Simple Audio Mixer API Overview

The simple audio mixer is an example of a TSSA component that has multiple inputs and one output. It also uses other TSSA compatible components to implement its function. In fact, the component mixes audio data. Because audio mixers can be called on to do so many things, they are necessarily application-specific. The simple audio mixer is provided in source form so that it can easily be customized to meet the needs of a user. This mixer is documented to guide a user into an implementation that takes advantages of the features of TSSA. As a simple mixer, the component requires that all packets be of the same size, and the same sample rate. It provides only volume scaling, and a simple filter.

The simple mixer is also an example of a "DVP audio mixer." Audio systems in the Philips Digital Video Platform (DVP) can be designed using this example as a template.

The simple mixer is implemented with separate AL and OL layers. It is demonstrated operating in asynchronous mode, as a task. Its implementation would allow it to be used in a synchronous (functional) mode, although this is not demonstrated. It is designed to be operated mainly from the OL layer.

The simple mixer is also designed to act as a reference for the creation of new TSSA components. It can serve as a guide to TSSA standard compliance. It is a reasonably complete TSSA module that accepts three audio inputs and mixes them together into one audio output. Because it does demonstrate a significant functionality, it may seem dauntingly large at first glance. CopyIO is a more simple example.

Since AmixSimple has both inputs and outputs, and since it is implemented using a pSOS task, it is an example of a task based filter.

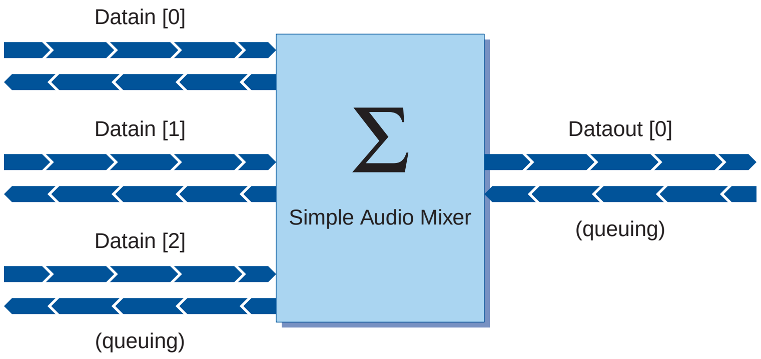


**Figure 4** Structure of the Simple Audio Mixer

## Background

This document assumes that the reader is familiar with the concepts of TSSA that are documented in Book 3, *Software Architecture*.

## The Files

Files making up the AmixSimple live in several directories:

- include/tmalAmixSimple.h

  The "TAS" include directory holds tmalAmixSimple.h. This file defines the bulk of the public interface. As the definition of the public interface, it contains only type definitions, preprocessor defines, and function prototypes for things that are to be accessed by users of the module. As an AL layer, it is designed to be usable whether or not a user desires the more sophisticated features of the OL layer. But the OL layer interface is defined in terms of this, the AL layer.

- include/tmolAmixSimple.h

  The "TAS" include directory also holds tmolAmixSimple.h. This is usually the primary public interface to the mixer. It includes the AL layer interface definition, and its structures are all defined in terms of the AL layer structures. The OL layer depends on an operating system, and so it is a higher level interface than the AL layer. The included example program uses the mixer from the OL layer.

- example/exolAmixSimple/exolAmixSimple.c

  The example directory contains a program that demonstrates the use of the mixer library. A user of the library will probably copy code out of the example program. The exol prefix tells us that this is an example of the use of the OL interface to the AmixSimple. Other files in this directory are the makefile and supporting files for pSOS.

- lib/AmixSimple/tmolAmixSimple.c

  This is the source for the OL layer of the AmixSimple. As you will see when you examine the OL layer source, it is rather generic. The OL layer uses the default functions to initialize the AL layer so that it can share the standard TSSA interface code. tmolAmixSimple.c is commented with "your code here" to point out places where your component will be different from the standard component. As explained in the TSSA software architecture document, the OL layer contains the operating system dependence, and it could be released to a customer without compromising the intellectual property contained in the component.

- lib/AmixSimple/tmalAmixSimple.c

  The AL layer source file defines the functions that make up the public interface to the AL layer. In some cases, the tmal file might contain code describing the entire module. In other cases, as in the mixer, the tmal file just defines the public interface. Other files are used to define the private portion of the interface. Sometimes a directory structure is used to organize the AL layer source. It can be arbitrarily complex.

The AL layer start function defines a loop that is the body of the component. Buffer management is implemented using the standard set of callbacks, as described in the software architecture manual.

- lib/AmixSimple/AmixSimpleCore.h

  The file filling this need is sometimes called “mix_private.h.” It holds definitions that are shared amoung the files of the mixer, but which are not part of the public interface. This file will not be included in example applications.

- lib/AmixSimple/AmixSimpleCore.c

  The core of the simple mixer is an example of a “DVP ready” audio mixer core. A DVP ready mixer core consists of “deinterleave” and “interleave” functions bracketing calls to “Audio Signal Processing (ASP)” components. The deinterleave function does the actual mixing of the three input channels, and its output is an array of non-interleaved “channel buffers.” The interleave function takes the array of channel buffers and creates an output packet in the correct format. This choice of architecture results from a number of considerations:

  — We want to be able to re-use the ASP components.

  — The input and output packets may be of several types, with various numbers of channels and various numbers of bits. The de-interleave and interleave functions handle all possible conversions to and from the 32 bit integer channel buffers that are used by the ASP functions.

  — The signal processing functions can make better use of the cache because the data is not interleaved.

  — Systems that require the output channels to be offset in time (like Dolby compliant speaker location controls) can use less memory, as the delay is specified on a “per channel” basis.

Because the ASP components have a simple, functional interface, it is easy to run the core algorithm on the TriMedia simulator. When the simulator is used in this way on the core of an algorithm, cycle-accurate execution numbers are easily obtained for convenient optimization.

See the TriMedia Software Architecture reference for more information about ASP components.

## The AspLpf Component

The AspLpf component is provided as a simple template of a typical signal processing component. You can notice that AspLpf exports the standard functions along with an instance setup structure, and that the include file for this is stored in the TAS include directory. Note also the use of the private instance variable.

## Simple Audio Mixer Inputs and Outputs

The simple audio mixer has three inputs and one output. Since the mixer is an example, rather strict limits are imposed on the formats of the input and output. A user should customize the mixer to support the formats that are really required.

The simple mixer accepts only stereo, 16-bit inputs and it generates only stereo, 16-bit output. All input packets are constrained to be the same size as the output packets.

The mixer includes a two pole low pass filter that is applied to the output stream. This is simply for demonstration purposes.

## Simple Audio Mixer Progress

The simple audio mixer does not require a progress function It uses the default progress function to handle changes of format.

## Simple Audio Mixer Errors

Errors can be reported during the mixer's setup phase, or at run time. Errors reported during the setup phase will be noticed as non-zero return values from the API. The definition of these constants is found in tmalAmixSimple.h. In addition, the library used in its debugging mode will use the assert mechanism to flag invalid inputs. These errors are covered along with the descriptions of each function in the API.

The simple audio mixer supports the installation of an error callback function. This function is invoked for a set of run time errors, as described below. None of the errors handled by the error callback function are considered fatal. The error function prototype is of the type **tsaErrorFunc_t**:

```
typedef tmLibappErr_t(*tsaErrorFunc_t)(Int instId, UInt32 flags,
ptsaErrorArgs_t args);
typedef struct tsaErrorArgs {
   Int           errorCode;
   Pointer       description;
} tsaErrorArgs_t, *ptsaErrorArgs_t;
```

Handlers should be provided for these possible values of the errorCode:

| | |
|---|---|
| `TMLIBAPP_ERR_UNDERRUN` | The mixer requested data but none was available. This error is not currently logged by the simple mixer. Its implementation is left to the user. |
| `AMIX_SIMPLE_ERR_IO_BUFFER_SIZE_MISMATCH` | Triggered when the dataSize of the received packet is not the same as **samplesPerPacket** (specified in the instance setup) and outChan.bytes- |

PerSample, computed from the initial format. The error description is an integer pointer.

```
description[0]:  channel ID. 256 for output channel
description[1]:  dataSize of offending packet
description[2]:  expected packet size.
description[3]:  ID of offending packet.
```

## Simple Audio Mixer Configuration

The simple audio mixer provides a queue-based configuration function. It can be used to change the volumes of the various channels and the filter cutoff frequency. The queue-based implementation is discussed in some depth in Book 3, *Software Architecture*.

```
   tmLibappErr_t tmolAmixSimpleInstanceConfig (
      Int instance,
      UInt32 flags,
      ptsaControlArgs_t args
   );
```

Values for the command entry in the args structure are defined in tmalAmixSimple.h, and are described below:

| | |
|---|---|
| `AMIX_SIMPLE_CONFIG_MASTER_VOLUME` | The output volume is adjusted. The parameter field of the control structure will contain an integer describing the volume in 1/100 dB. Positive values provide gain, negative values attenuation. |
| `AMIX_SIMPLE_CONFIG_CHANNEL_VOLUME` | The input channel volume is adjusted. The parameter field of the control structure is treated as a pointer to an array of integers. The first member is the channel ID. The second is an integer describing the volume in 0.01 dB. Positive values provide gain, negative values attenuation. |
| `AMIX_SIMPLE_CONFIG_LPF_FREQUENCY` | The output volume is adjusted. The parameter field of the control structure will contain an integer describing the volume in 0.01 dB. Positive values provide gain, negative values attenuation. |
| `AMIX_SIMPLE_CONFIG_INPUT` | Not currently implemented. |

## Simple Audio Mixer API Data Structures

This section presents the data structures contained in the Simple Audio Mixer library.

## tmolAmixSimpleCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmolAmixSimpleCapabilities_t; *ptmolAmixSimpleCapabilities_t;
```

### Fields

| | |
|---|---|
| `defaultCapabilities` | For compliance with the application library architecture, this is a pointer to a structure of the standard type. |

### Description

**The tmolAmixSimpleCapabilities_t** structure describes the capabilities and requirements of the simple audio mixer module. A user can retrieve the structure's address by calling **tmolAmixSimpleGetCapabilities**.

## tmolAmixSimpleInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t   defaultSetup;
   Int                          masterVolume;
   Int                          channelVolume
                                [AMIX_SIMPLE_NUM_INPUT_PORTS];
   Int                          lpfFrequency;
   Int                          samplesPerPacket;
} tmolAmixSimpleInstanceSetup_t; *ptmolAmixSimpleInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `defaultSetup` | For compliance with TSA, this is a pointer to a structure of the standard type. |
| `masterVolume` | An integer specifying output volume in 0.01 dB. |
| `channelVolume` | An array of integers, one per input channel, specifying input volume in 0.01 dB. |
| `lpfFrequency` | A two-pole low-pass filter is applied to the output of the mixer. Its cutoff frequency is specified in Hertz. |
| `samplesPerPacket` | Since all packets must have the same size, it is specified here. |

### Description

**The tmolAmixSimpleInstanceSetup_t** structure is used to describe the initial operation of this instance of the mixer.

## Simple Audio Mixer API Functions

This section presents the functions contained in the Simple Audio Mixer library.

## tmolAmixSimpleGetCapabilities

```
tmLibappErr_t tmolAmixSimpleGetCapabilities(
   ptmolAmixSimpleCapabilities_t   *pCap
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return a pointer to capabilities data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Retrieves the capabilites of the simple audio mixer. The function pointer that is returned remains valid as long as the digitizer is active.

## tmolAmixSimpleOpen

```
tmLibappErr_t tmolAmixSimpleOpen(
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | No more instances of the simple audio mixer are available. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Memory allocation failed. |

### Description

The open function creates an instance of the simple audio mixer and informs the user of its instance. The simple audio mixer supports only one instance.

## tmolAmixSimpleGetInstanceSetup

```
tmLibappErr_t tmolAmixSimpleGetInstanceSetup(
   Int                              instance,
   ptmolAmixSimpleInstanceSetup_t  *setup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmolAmixSimpleOpen**. |
| setup | Pointer to a variable in which to return a pointer to the setup data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

A pointer to the current instance setup is retrieved. After a call to **tmolAmixSimpleOpen**, this structure is filled with default values to simplify the impending call to **tmolAmix-SimpleInstanceSetup**.

## tmolAmixSimpleInstanceSetup

```
tmLibappErr_t tmolAmixSimpleInstanceSetup(
   Int                              instance,
   ptmolAmixSimpleInstanceSetup_t   setup
);
```

### Parameters

| | |
|---|---|
| `instance` | As returned from **tmolAmixSimpleOpen**. |
| `setup` | Pointer to setup data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Specified instance does not match current instance. Digitizer supports only one instance. |
| `TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE` | An unsupported data format was requested. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Memory allocation failed. |

Other errors are possibly reported by the device library or board support package.

### Description

The simple audio mixer is prepared for operation. Parameters are checked. The mixer is left "stopped." It will become operational on a call to **tmolAmixSimpleStart**.

## tmolAmixSimpleStart

```
tmLibappErr_t tmolAmixSimpleStart(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmolAmixSimpleOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

The mixer represented by the instance is started.

## tmolAmixSimpleStop

```
tmLibappErr_t tmolAmixSimpleStop(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmolAmixSimpleOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

The mixer represented by the instance is stopped.

## tmolAmixSimpleInstanceConfig

```
tmLibappErr_t tmolAmixSimpleInstanceConfig(
   Int                instance,
   UInt32             flags,
   ptsaControlArgs_t  args
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmolAmixSimpleOpen**. |
| `flags` | Not used by **tmolAmixSimpleInstanceConfig**. |
| `args` | Points to a control structure used to modify the operation of the simple audio mixer. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

Errors detected by the underlying **tmalAmixSimple** call can be found in the retval member of the control structure.

### Description

While a module is operating, the configuration function can be used to change the operating parameters. In the example, a few simple operations are implemented.

| | |
|---|---|
| `AMIX_SIMPLE_CONFIG_MASTER_VOLUME` | The output volume is adjusted. The parameter field of the control structure will contain an integer describing the volume in 0.01 dB. Positive values provide gain, negative values attenuation. |
| `AMIX_SIMPLE_CONFIG_CHANNEL_VOLUME` | The input channel volume is adjusted. The parameter field of the control structure is treated as a pointer to an array of integers. The first member is the channel ID. The second is an integer describing the volume in 0.01 dB. Positive values provide gain, negative values attenuation. |
| `AMIX_SIMPLE_CONFIG_LPF_FREQUENCY` | The output volume is adjusted. The parameter field of the control structure will contain an integer describing the volume in 1/100 dB. Positive values provide gain, negative values attenuation. |
| `AMIX_SIMPLE_CONFIG_INPUT` | Not currently implemented. |

# Chapter 9

# Noise Sequencer (NoiseSeq) API

**Note**
This component library is not included with the basic TriMedia SDE, but is available as a part of other software packages, under a separate licensing agreement. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

## Introduction

The TriMedia Noise Sequencer library can be used to produce a calibration signal for multichannel audio systems. It produces pink noise at frequencies of 32, 44.1 and 48 kHz. The library can be used at both the AL layer and the OL layer. It provides a flexible interface with several different configuration possibilities.

The generated pink noise is produced on one output channel at a time. All other output channels are muted. It is possible to automatically change the output channel after a user has specified channel duration time. In the automatic channel switch mode, the noise source can either be moved in clockwise or counter-clockwise direction. A channel mask determines the channels on which the noise will be presented.

All options mentioned above can be applied statically as well as dynamically. The static configuration is performed by **tmXlNoiseSeqInstanceSetup** and the dynamic configuration by **tmXlNoiseSeqInstanceConfig**.

One example program is provided with this library that shows how an OL layer application of the Noise Sequencer can be implemented. The program *exalNoiseSeq.c* connects the Noise Sequencer with the Audio Renderer. It shows how dynamic configuration changes can be performed with simple commands.

### Spectrum of the Pink Noise

The following plot illustrates the spectrum of the pink noise which is produced by the TriMedia Noise Sequencer library.

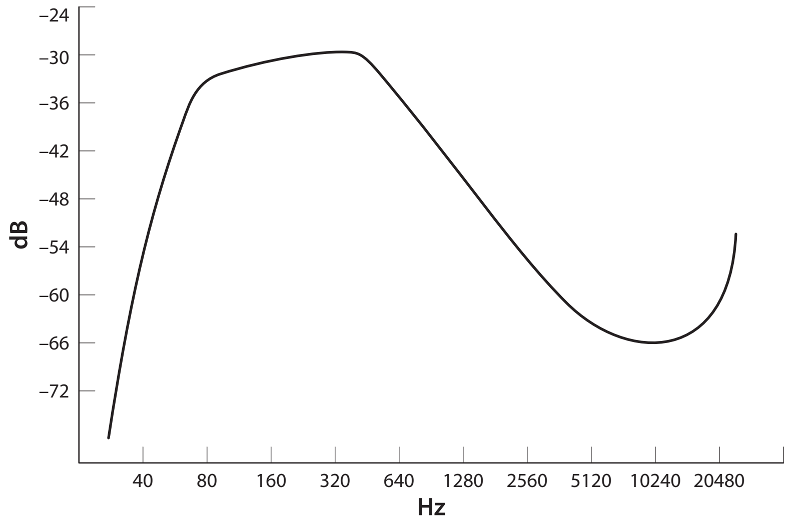


**Figure 5** Spectrum of the colored noise (44100 Hz sampling rate)

## Noise Sequencer Inputs and Outputs

The TriMedia Noise Sequencer is a component that provides one output and no input pins. It supports only one 16-bit PCM output format which is **apfFiveDotOne16**. As sampling frequencies only 32, 44.1, and 48 kHz are supported.

## Noise Sequencer Errors

The error callback function can convey the following error messages:

| | |
|---|---|
| `TMLIBAPP_ERR_ALREADY_STARTED` | The instance of the decoder is already started. |
| `NS_ERR_ILL_DATASIZE` | The size of the current output packet is too small. It must be big enough to accommodate at least on multichannel sample, which is 12 bytes for **apfFiveDotOn16**. |

## Noise Sequencer Progress

The Noise Sequencer does not use the progress callback function except for installing an output format. Refer to the respective chapter in the TSSA documentation to find more details on this specific usage of the progress function.

## Noise Sequencer Configuration

The Noise Sequencer API provides two configuration functions, one for the usage at the AL layer and one for the OL layer. The AL layer function **tmalNoiseSeqInstanceConfig** executes one of the below described commands in the classical fashion directly upon its call. The OL layer function **tmolNoiseSeqInstanceConfic** on the other hand uses its AL layer counterpart to perform the desired action. It uses either a control queue to communicate with the AL layer function, or it can call it directly. It does the latter if **tmolNoiseSeqInstanceConfig** is called in the context of the Noise Sequencer start function (same task).

Both config function have a pointer to a struct of the type **tsaControlArgs_t** as parameter which consists of four fields:

```
typedef struct tsaControlArgs {
   UInt32        command;
   Pointer       parameter;
   tmLibappErr_t retval;
   UInt32        timeout;
} tsaControlArgs_t, *ptsaControlArgs_t;
```

The application writes one of the below described commands into the **command** field. The **parameter** field is used to either send (**_SET_** commands) a value to the AC-3 decoder or receive (**_GET_** commands) a value from it. In both cases type casts must be applied in the application because the **parameter** field contains a void pointer. In some cases the **parameter** field is not used.

The remaining two fields are not used by **tmalNoiseSeqInstanceConfig**. Its OL layer counterpart is using the **timeout** value for the access to the control queue. It is measured in pSOS+ clock ticks. If its value is zero it waits forever. The **retval** field is filled by **tmolNoiseSeqInstanceConfig** with the return value of **tmalNoiseSeqInstanceConfig**. The application must check this value as well as the return value of **tmolNoiseSeqInstanceConfig** which indicates problems with the control queue.

An OL layer application must provide a control descriptor during the initialization phase (before calling **tmolNoiseSeqInstanceSetup**). The queues of this descriptor carry the commands between the application and the Noise Sequencer component.

**Table 1** Commands to Change the Configuration

| Command | Supported Values for Parameter |
|---|---|
| NS_CONFIG_SET_OUTCHAN | all values defined in **tmalNoiseSeqOutChan_t**, except for **NS_OUTCHAN_NONE** |
| NS_CONFIG_SET_NEXT_OUTCHAN | no parameter required |
| NS_CONFIG_SET_OUTCHAN_MASK | parameter contains the channels on which noise is to be rendered, values of **tmalNoiseSeqOutChan_t** are OR'd for this purpose |
| NS_CONFIG_SET_AUTOSWITCH_ON | no parameter required |
| NS_CONFIG_SET_AUTOSWITCH_OFF | no parameter required |
| NS_CONFIG_SET_SWITCHTIME | values greater or equal 0.1 |
| NS_CONFIG_SET_DIRECTION_CW | no parameter required |
| NS_CONFIG_SET_DIRECTION_ACW | no parameter required |

**Table 2** Commands to get Settings of Current Configuration

| Command | Parameter needs cast to |
|---|---|
| NS_CONFIG_GET_OUTCHAN | tmalNoiseSeqOutChan_t |
| NS_CONFIG_GET_OUTCHAN_MASK | Int32 |
| NS_CONFIG_GET_AUTOSWITCH_MODE | Bool |
| NS_CONFIG_GET_SWITCHTIME | Float |
| NS_CONFIG_GET_DIRECTION | tmalNoiseSeqDirection_t |

If a floating point value is to be sent to the Noise Sequencer or to be received by the application, a special casting mechanism is required. The switch time can be set to 0.8 with the following operations:

```
tsaControlArgs_t cargs;
Float32    fval = 0.8;
cargs.command   = NS_CONFIG_SET_SWITCHTIME;
cargs.parameter = *((Pointer *) &fval);
tmolNoiseSeqInstanceConfig( decInstance, tsaControlWait, &cargs);
```

If the application wants to obtain the current setting of this Noise Sequencer parameter it must do the following:

```
tsaControlArgs_t cargs;
Float32          fval;
cargs.command   = NS_CONFIG_GET_SWITCHTIME;
tmolNoiseSeqInstanceConfig( decInstance, tsaControlWait, &cargs);
fval = *((Float32 *) &cargs.parameter);
```

This casting is required because an implicit cast to integer would otherwise be performed by the compiler. For more details on what actions the commands perform, see page 204.

**Note**
If **tmolNoiseSeqInstanceConfig** is called within the context of the Noise Sequencer, the command gets executed immediately. This is the case when this function gets called from a callback function. The queue mechanism is used only when the function call happens in a separate task.

## Noise Sequencer AL Layer API Data Structures

This section presents the Noise Sequencer API data structures.

| Name | Page |
|---|---|
| tmalNoiseSeqCapabilities_t | 202 |
| tmalNoiseSeqInstanceSetup_t | 203 |
| tmalNoiseSeqCommands_t | 204 |
| tmalNoiseSeqOutChan_t | 206 |
| tmalNoiseSeqDirection_t | 207 |

## tmalNoiseSeqCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmalNoiseSeqCapabilities_t, *ptmalNoiseSeqCapabilities_t;
```

### Fields

| | |
|---|---|
| `defaultCapabilities` | Pointer to default capabilities structure. |

### Description

A pointer to a struct of this type is returned by **tmalNoiseSeqGetCapabilities**. It contains information about the properties of the Noise Sequencer library.

## tmalNoiseSeqInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   tmalNoiseSeqOutChan_t       outChan;
   Int32                       outChanMask;
   Bool                        autoChanSwitch;
   Float                       switchTime;
   tmalNoiseSeqDirection_t     switchDir;
} tmalNoiseSeqInstanceSetup_t, *ptmalNoiseSeqInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `defaultSetup` | Pointer to default instance setup struct (see tsa.h). |
| `outChan` | Audio channel on which the noise will appear first. |
| `outChanMask` | Bit mask of the output channels on which the noise should be produced. Active channels are selected by ORing **tmalNoiseSeqOutChan_t** values. |
| `autoChanSwitch` | If True, automatic channel switching is performed. If False, noise will only be produced on **outChan**. |
| `switchTime` | Time in seconds after which the channel swapping is performed, if **autoChanSwitch** is True. Minimum value is 0.1. |
| `switchDir` | Determines the direction the noise moves from speaker to speaker. |

### Description

A struct of this type is used to configure the noise sequencer prior to starting the data streaming. The configuration of the component is performed with **tmalNoiseSeqInstanceSetup**.

## tmalNoiseSeqCommands_t

```
typedef enum {
   NS_CONFIG_SET_OUTCHAN         = tsaCmdUserBase + Øx00,
   NS_CONFIG_GET_OUTCHAN         = tsaCmdUserBase + Øx01,
   NS_CONFIG_SET_NEXT_OUTCHAN    = tsaCmdUserBase + Øx02,
   NS_CONFIG_SET_OUTCHAN_MASK    = tsaCmdUserBase + Øx03,
   NS_CONFIG_GET_OUTCHAN_MASK    = tsaCmdUserBase + Øx04,
   NS_CONFIG_SET_AUTOSWITCH_ON   = tsaCmdUserBase + Øx05,
   NS_CONFIG_SET_AUTOSWITCH_OFF  = tsaCmdUserBase + Øx06,
   NS_CONFIG_GET_AUTOSWITCH_MODE = tsaCmdUserBase + Øx07,
   NS_CONFIG_SET_SWITCHTIME      = tsaCmdUserBase + Øx08,
   NS_CONFIG_GET_SWITCHTIME      = tsaCmdUserBase + Øx09,
   NS_CONFIG_SET_DIRECTION_CW    = tsaCmdUserBase + Øx0a,
   NS_CONFIG_SET_DIRECTION_ACW   = tsaCmdUserBase + Øx0b,
   NS_CONFIG_GET_DIRECTION       = tsaCmdUserBase + Øx0c
} tmalNoiseSeqCommands_t;
```

### Fields

| | |
|---|---|
| `NS_CONFIG_SET_OUTCHAN` | Changes the output channel to channel stored in parameter field of the control arguments struct, see **tmalNoiseSeqOutChan_t** for supported values. |
| `NS_CONFIG_GET_OUTCHAN` | Returns the current output channel in the parameter field. |
| `NS_CONFIG_SET_NEXT_OUTCHAN` | Switches the noise to the next active channel in the specified rotation direction, no parameter is required. |
| `NS_CONFIG_SET_OUTCHAN_MASK` | Sets a new mask for the active output channels |
| `NS_CONFIG_GET_OUTCHAN_MASK` | Returns the current output channel mask |
| `NS_CONFIG_SET_AUTOSWITCH_ON` | Enables the automatic switching of the output channel in clockwise order. |
| `NS_CONFIG_SET_AUTOSWITCH_OFF` | Disables the automatic switching of the output channel. |
| `NS_CONFIG_GET_AUTOSWITCH_MODE` | Returns a boolean in the parameter field indicating if automatic channel switching is enabled or disabled. |
| `NS_CONFIG_SET_SWITCHTIME` | Sets the time after which the channels are switched, value is stored in the parameter field. It is of the type float and may not less 0.1. |
| `NS_CONFIG_GET_SWITCHTIME` | Returns the current switch time in the parameter field. |
| `NS_CONFIG_SET_DIRECTION_CW` | Sets the noise rotation direction to clockwise. |
| `NS_CONFIG_SET_DIRECTION_ACW` | Sets the noise rotation direction to counter-clockwise. |

| | |
|---|---|
| `NS_CONFIG_GET_DIRECTION` | Gets the noise rotation direction. |

## Description

Specifies supported commands for **tmalNoiseSeqInstanceConfig**.

## tmalNoiseSeqOutChan_t

```
typedef enum {
   NS_OUTCHAN_NONE      = 0x00000000,
   NS_OUTCHAN_LEFT      = 0x00000001,
   NS_OUTCHAN_RIGHT     = 0x00000002,
   NS_OUTCHAN_CENTER    = 0x00000004,
   NS_OUTCHAN_LEFTSUR   = 0x00000008,
   NS_OUTCHAN_RIGHTSUR  = 0x00000010
} tmalNoiseSeqOutChan_t;
```

### Fields

| | |
|---|---|
| `NS_OUTCHAN_NONE` | This value must not be used. |
| `NS_OUTCHAN_LEFT` | Left channel is output channel for the pink noise. |
| `NS_OUTCHAN_RIGHT` | Right channel is output channel for the pink noise. |
| `NS_OUTCHAN_CENTER` | Center channel is output channel for the pink noise. |
| `NS_OUTCHAN_LEFTSUR` | Left surround channel is output channel for the pink noise. |
| `NS_OUTCHAN_RIGHTSUR` | Right surround channel is output channel for the pink noise. |

### Description

Specifies supported output channels for the noise.

## tmalNoiseSeqDirection_t

```
typedef enum {
NS_DIRECTION_CW   = 0x00000001,
NS_DIRECTION_ACW  = 0x00000002
} tmalNoiseSeqDirection_t;
```

### Fields

| | |
|---|---|
| `NS_DIRECTION_CW` | Noise moves in clockwise direction (left, center, right, rsur, lsur). |
| `NS_DIRECTION_ACW` | Noise moves in counter-clockwise direction (left, lsur, rsur, right, center). |

### Description

Used to determine the steering direction of the noise.

## Noise Sequencer AL Layer API Functions

This section describes the Noise Sequencer API functions.

| Name | Page |
|---|---|
| tmalNoiseSeqGetCapabilities | 209 |
| tmalNoiseSeqOpen | 210 |
| tmalNoiseSeqClose | 211 |
| tmalNoiseSeqInstanceSetup | 212 |
| tmalNoiseSeqStart | 213 |
| tmalNoiseSeqStop | 214 |
| tmalNoiseSeqInstanceConfig | 215 |

## tmalNoiseSeqGetCapabilities

```
extern tmLibappErr_t tmalNoiseSeqGetCapabilities  (
   ptmalNoiseSeqCapabilities_t   *cap
);
```

### Parameters

| | |
|---|---|
| `cap` | Pointer to **tmalNoiseSeqCapabilities_t**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Returns a pointer to the capabilities of the Noise Sequencer.

## tmalNoiseSeqOpen

```
extern tmLibappErr_t tmalNoiseSeqOpen (
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer to the instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | If memory allocation for the instance variables failed. |

### Description

Assigns an instance of the Noise Sequencer for usage.

## tmalNoiseSeqClose

```
extern tmLibappErr_t tmalNoiseSeqClose (
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Number of the instance to be closed. |

### Return Codes

| | |
|---|---|
| TMLIBAPP_OK | Success. |
| TMLIBAPP_ERR_INVALID_INSTANCE | If instance is not a valid instance. |

### Description

De-assigns instance for usage. Requires **tmalNoiseSeqOpen** to be called first.

## tmalNoiseSeqInstanceSetup

```
extern tmLibappErr_t tmalNoiseSeqInstanceSetup (
   Int                            instance,
   tmalNoiseSeqInstanceSetup_t  *setup
);
```

### Parameters

| | |
|---|---|
| `instance` | The instance. |
| `setup` | Pointer to the NS setup structure **tmalNoiseSeq-InstanceSetup_t**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | If instance is not a valid instance. |
| `TMLIBAPP_ERR_INVALID_SETUP` | If setup parameters are not valid (**parentId**, **dataoutFunc** or **completionFunc** are missing). |
| `TMLIBAPP_ERR_NOT_STOPPED` | If the instance was not in the stopped state. |
| `NS_ERR_ILL_OUT_CHAN` | If the **outChan** field of the setup structure contains an invalid value. |
| `NS_ERR_ILL_TIME_CONST` | If the switchTime field of the setup structure contains an invalid value (less than 0.1). |
| `NS_ERR_ILL_FREQ` | If the sampling frequency specified in the output descriptor is not supported, supported values are 48K, 44K, and 32K. |
| `NS_ERR_ILL_DIR` | If specified steering direction is not defined in **tmalNoiseSeqDirection_t**. |
| `NS_ERR_ILL_CHAN_MASK` | Channel mask is either zero or contains values not defined in **tmalNoiseSeqOutChan_t**. |
| `TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE` | If the specified audio dataSubtype is not supported. Currently only **apfFiveDotOne16** is supported. |

### Description

Sets up the instance of the Noise Sequencer. Requires **tmalNoiseSeqOpen** to be called first. Instance must be stopped (**Open** automatically changes state to stopped).

## tmalNoiseSeqStart

```
extern tmLibappErr_t tmalNoiseSeqStart (
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | The instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | If instance is not a valid instance. |
| `TMLIBAPP_ERR_NOT_SETUP` | If instance has not been setup previously. |
| `TMLIBAPP_ERR_ALREADY_STARTED` | If instance has already been started. |
| `NS_ERR_ILL_DATASIZE` | If **bufSize** of empty output packet is too small, (less than one sample across all channels). |

### Description

This function starts the data streaming.

## tmalNoiseSeqStop

```
extern tmLibappErr_t tmalNoiseSeqStop (
   Int  instance
);
```

### Parameters

| | |
|---|---|
| `instance` | The instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | If instance is not a valid instance. |
| `TMLIBAPP_ERR_NOT_SETUP` | If instance has not been setup previously. |
| `TMLIBAPP_ERR_ALREADY_STOPPED` | If instance has already been stopped. |

### Description

Stops data streaming.

## tmalNoiseSeqInstanceConfig

```
extern tmLibappErr_t tmalNoiseSeqInstanceConfig  (
   Int                 instance,
   ptsaControlArgs_t  cmdDesc
);
```

### Parameters

| | |
|---|---|
| `Instance` | The instance. |
| `cmdDesc` | Pointer to control parameter struct containing a command and a parameter. The usage of the parameter depends on the command. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | If instance is not a valid instance. |
| `TMLIBAPP_ERR_NOT_SETUP` | If instance has not been setup previously. |
| `FR_ERR_UNKNOWN_COMMAND` | If the supplied command is not a valid one. |
| `NS_ERR_ILL_OUT_CHAN` | If selected output channel is not supported or not marked active in the output channel bit mask. |
| `NS_ERR_ILL_TIME_CONST` | If the switch time value in the parameter field is less 0.1. |

### Description

Effects a command on an open instance. See **tmalNoiseSeqCommands_t** for the supported commands.

## Noise Sequencer OL Layer API Data Structures

This section presents the Noise Sequencer API data structures specific to the OL API.

| Name | Page |
|---|---|
| tmolNoiseSeqCapabilities_t | 217 |
| tmolNoiseSeqInstanceSetup_t | 218 |

## tmolNoiseSeqCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmolNoiseSeqCapabilities_t, *ptmolNoiseSeqCapabilities_t;
```

### Fields

| | |
|---|---|
| `defaultCapabilities` | Pointer to default capabilities structure. |

### Description

A pointer to a structure of this type is returned by **tmolNoiseSeqGetCapabilities**. It contains information about the properties of the Noise Sequencer library.

## tmolNoiseSeqInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   tmalNoiseSeqOutChan_t       outChan;
   Int32                       outChanMask;
   Bool                        autoChanSwitch;
   Float                       switchTime;
   tmalNoiseSeqDirection_t     switchDir;
} tmolNoiseSeqInstanceSetup_t, *ptmolNoiseSeqInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `defaultSetup` | Pointer to default instance setup struct (see tsa.h). |
| `outChan` | Audio channel on which the noise will appear first. |
| `outChanMask` | Bit mask of the output channels on which the noise should be produced. Active channels are selected by ORing **tmalNoiseSeqOutChan_t** values. |
| `autoChanSwitch` | If True, automatic channel switching is performed. If False, noise will only be produced on **outChan**. |
| `switchTime` | Time in seconds after which the channel switching is performed, if **autoChanSwitch** is True. Minimum value is 0.1. |
| `switchDir` | Determines the direction the noise moves from speaker to speaker. |

### Description

A struct of this type is used to configure the noise sequencer prior to starting the data streaming. A pointer to a pre-configured instance setup struct can be obtained by **tmolNoiseSeqGetInstanceSetup**. The configuration of the Noise Sequencer component is performed with **tmolNoseSeqInstanceSetup**.

## Noise Sequencer OL Layer API Functions

This section presents the Noise Sequencer OL functions.

| Name | Page |
|---|---|
| tmolNoiseSeqGetCapabilities | 220 |
| tmolNoiseSeqOpen | 220 |
| tmolNoiseSeqClose | 221 |
| tmolNoiseSeqGetInstanceSetup | 222 |
| tmolNoiseSeqInstanceSetup | 223 |
| tmolNoiseSeqStart | 224 |
| tmolNoiseSeqStop | 225 |
| tmolNoiseSeqInstanceConfig | 226 |

## tmolNoiseSeqGetCapabilities

```
extern tmLibappErr_t tmolNoiseSeqGetCapabilities  (
   ptmolNoiseSeqCapabilities_t   *cap
);
```

### Parameters

| | |
|---|---|
| `cap` | Pointer to a variable in which to return a pointer to capabilities data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Returns a pointer to the capabilities of the Noise Sequencer.

## tmolNoiseSeqOpen

```
extern tmLibappErr_t tmolNoiseSeqOpen (
   Int  *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Memory allocation failed. |

### Description

Assigns an instance of the Noise Sequencer for usage.

## tmolNoiseSeqClose

```
extern tmLibappErr_t tmolNoiseSeqClose (
   Int  instance
);
```

### Parameters

| | |
|---|---|
| `instance` | The instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Asserts

| | |
|---|---|
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Not a valid instance. |

### Description

De-assigns instance for usage. Requires **tmolNoiseSeqOpen** to be called first.

## tmolNoiseSeqGetInstanceSetup

```
extern tmLibappErr_t tmolNoiseSeqGetInstanceSetup (
   Int                            instance
   ptmolNoiseSeqInstanceSetup_t  *setup
);
```

### Parameters

| | |
|---|---|
| `instance` | The instance. |
| `setup` | Pointer to a variable in which to return a pointer to the setup data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Asserts

| | |
|---|---|
| **TMLIBAPP_ERR_INVALID_INSTANCE** | If instance is not a valid instance. |

### Description

This function can be used to obtain a pointer to a pre-configured and allocated instance setup struct. The memory for this struct is allocated by **tmolNoiseSeqOpen**. Requires **tmolNoiseSeqOpen** to be called first.

## tmolNoiseSeqInstanceSetup

```
extern tmLibappErr_t tmolNoiseSeqInstanceSetup (
   Int                            instance,
   tmolNoiseSeqInstanceSetup_t  *setup
);
```

### Parameters

| | |
|---|---|
| `instance` | The instance. |
| `setup` | Pointer to the setup structure. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | If instance is not a valid instance. |
| `TMLIBAPP_ERR_INVALID_SETUP` | If setup parameters are not valid (**parentId**, **dataoutFunc** or **completionFunc** are missing). |
| `TMLIBAPP_ERR_NOT_STOPPED` | If the instance was not in the stopped state. |
| `NS_ERR_ILL_OUT_CHAN` | If the **outChan** field of the setup structure contains an invalid value. |
| `NS_ERR_ILL_TIME_CONST` | If the switchTime field of the setup structure contains an invalid value (< 0.1). |
| `NS_ERR_ILL_FREQ` | If the sampling frequency specified in the output descriptor is not supported, supported values are 48K, 44K, and 32K. |
| `NS_ERR_ILL_DIR` | If specified steering direction is not defined in **tmalNoiseSeqDirection_t**. |
| `NS_ERR_ILL_CHAN_MASK` | Channel mask is either zero or contains values not defined in **tmalNoiseSeqOutChan_t**. |
| `TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE` | If the specified audio **dataSubtype** is not supported. Currently only **apfFiveDotOne16** is supported. |

### Asserts

| | |
|---|---|
| `TMLIBAPP_ERR_INVALID_INSTANCE` | If instance is not a valid instance. |
| `TMLIBAPP_ERR_INVALID_SETUP` | If setup pointer is a NULL pointer. |

### Description

Sets up the instance of the Noise Sequencer. Requires **tmolNoiseSeqOpen** to be called first. Instance must be stopped (**Open** automatically sets state to stopped).

## tmolNoiseSeqStart

```
extern tmLibappErr_t tmolNoiseSeqStart (
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | The instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_TCREATE_FAILED` | The creation of the task for the AL layer start function failed. |
| `TMLIBAPP_ERR_TSTART_FAILED` | The start of the task for the AL layer start function failed. |

### Asserts

| | |
|---|---|
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Not a valid instance. |
| `TMLIBAPP_ERR_NOT_SETUP` | Instance is not set up properly. |

### Description

Starts the AL layer start function as task, data streaming begins.

## tmolNoiseSeqStop

```
extern tmLibappErr_t tmolNoiseSeqStop (
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | The instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_TSUSPEND_FAILED` | The Noise Sequencer task could not be suspended. |

### Asserts

| | |
|---|---|
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Not a valid instance. |
| `TMLIBAPP_ERR_NOT_SETUP` | Instance is not set up properly. |

### Description

Stops data streaming.

## tmolNoiseSeqInstanceConfig

```
extern tmLibappErr_t tmolNoiseSeqInstanceConfig (
   Int                 instance,
   ptsaControlArgs_t   cmdDesc
);
```

### Parameters

| | |
|---|---|
| `Instance` | The instance. |
| `cmdDesc` | Pointer to control parameter struct containing a command and a parameter. The usage of the parameter depends on the command. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Asserts

| | |
|---|---|
| `TMLIBAPP_ERR_INVALID_INSTANCE` | If instance is not a valid instance. |
| `TMLIBAPP_ERR_NOT_SETUP` | If instance is not set up properly. |

### Description

Effects a command on an open instance. See page 204 for the supported commands and page 199 for the usage of the control struct and the relationship to **tmalNoiseSeq-InstanceConfig**.

Note that this function is just a communication interface to **tmalNoiseSeqInstance-Config**. If an error occurs during the execution of a command, the respective error massage is stored in the **retval** field of the struct to which **cmdDesc** points.

Chapter 10

# DTV Audio Mixer (AmixDtv) API

**Note**
This component library is not included with the basic TriMedia SDE, but is available as a part of other software packages, under a separate licensing agreement. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

## DTV Audio Mixer Overview

The DTV audio mixer is designed to provide convenient access to audio output for the various audio sources that are used in a digital television system. The DTV mixer does mix several audio streams into one output stream. But the DTV mixer contains a number of features that are designed specifically for digital TV. These include bass redirection and delay for the center and surround channels.

The DTV mixer supports three input channels. One of these is designed to be a six channel (5.1) stream. The output is usually an eight channel stream. The output can be accurate to 20 bits.

The DTV mixer is implemented as a TSSA module, and as such, it has an AL and an OL layer. Because it is tested only as an OL layer module, the AL layer functions are declared in the single OL layer include file. Only the OL layer is documented. The DTV mixer is an example of a task based filter with multiple inputs and one output.

### Background

This document assumes that the reader is familiar with the concepts of TSSA as documented in Book 3, *Software Architecture*. Some of the concepts like bass redirection are described in the Dolby Licensee Information Manual, V2.0. This is available from Dolby Labs.

## DTV Audio Mixer Inputs and Outputs

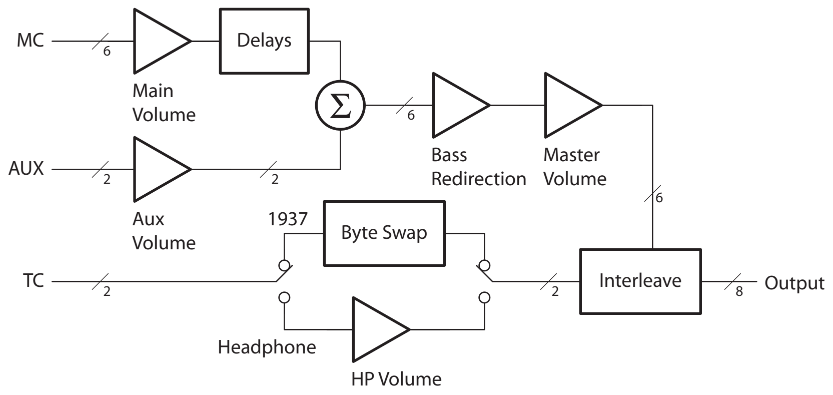


**Figure 6** Block Diagram of Mixer

The DTV audio mixer has three inputs and one output. Each of the inputs is specialized and named for its intended task. The formats that are supported are specified in the capabilities structure. The format of each channel in operation is determined using the format mechanism of TSSA.

- The MultiChannel (MC) input accepts stereo or six channel streams that are 16 or 20 bits wide. This stream is gain scaled, delayed, filtered, and redirected as specified by the user. At the end of this processing, the six channels are presented as six of the eight channels available on the output.

```
static tmAvFormat_t amixMultiChannelFormat = {
   ...
   avdcAudio,                                        /* dataClass      */
   atfLinearPCM,                                     /* dataType       */
   apfFiveDotOne16 | apfStereo16 | apfFiveDotOne32 | apfStereo32,
                                                     /* dataSubtype    */
   20                                                /* description    */
};
```

- The TwoChannel (TC) input accepts stereo 16 bit PCM data, or alternatively, IEC61937 formatted data (AC-3). It is also known as the “headphone” channel. The two channels of PCM can be gain scaled to provide a headphone output. The two channels of IEC61937 data are not gain scaled, but they are byte swapped. In any case, they are interleaved into the eight channel output stream and presented as the last pair of channels in the output.

```
static tmAvFormat_t  amixTwoChannelFormat = {
   ...                              /* data for AC3 or headphone */
   avdcAudio,                       /* dataClass      */
   atf1937 | atfLinearPCM,          /* dataType       */
   apfStereo16,                     /* dataSubtype    */
   16                               /* description    */
};
```

- The auxiliary channel (AUX) is not yet used. It is designed to support a second stereo stream that is mixed into the multi-channel output. It has not been tested.

```
static tmAvFormat_t  amixAuxFormat = {
   ...
   avdcAudio,           /* dataClass      */
   atfLinearPCM,        /* dataType       */
   apfStereo16,         /* dataSubtype    */
   16                   /* description    */
};
```

- The output is usually an eight channel stream, although a six channel mode is also supported for testing. The output is designed to interface directly with the audio renderer.

```
static tmAvFormat_t  amixOutputChannelFormat = {
   ...
   avdcAudio,                                        /* dataClass      */
   atfLinearPCM,                                     /* dataType       */
   apfFiveDotOne16 | apfSevenDotOne16 | apfFiveDotOne32 |
   apfSevenDotOne32,                                 /* dataSubtype    */
   20                                                /* description    */
};
```

All of these I/Os are fully TSSA compliant. In order to simplify the processing inside of the mixer, input packets are constrained to contain the same number of samples as output packets. When this is not the case, errors are reported using the error callback function.

## Operation of the DTV Audio Mixer

In simplified form, the buffer management loop of the DTV mixer is structured like this:

```
while (1) {
   getEmptyOutput packet (block here, and check the control queue);
   get full packet for each input channel
      (wait one tick at each valid input to allow scheduling)
   mix input channels to create output packet()
   send back empty input packets()
   send on full output packet()
}
```

The mixer copies the timestamp of the MC input packet to the output packet. The gain taken in the mixer is in two phases. In the preamp phase, gain or attenuation is taken using controls for each individual channel. The gain is limited to +12db, and the signal is saturated at this level. The output gain phase is combined with the bass redirection processing. A master gain (or attenuation) is applied to the summed output, and trim is applied. The gain applied at this stage is again limited to +12db, with the output signal saturated at full scale. In a product, positive gain is not usually applied to the Dolby Digital signal in the digital domain, as this will result in signal distortion. The decision to implement attenuation in the digital or analog domain depends on the structure of the D/A converters and the analog output system. All gain scaling has traditionally been implemented in the analog domain.

### Prime Buffer

In order to assure clean startup from cases where the input pipeline begins empty, the mixer can be configured to hold a configurable number of MC input buffers before starting actual operation. This is implemented as a loop preceding the main processing loop, as described above. To ensure that the audio system passes the Dolby “start/stop” test from an SPDIF input, it has been found that 19 buffers are required to prime the mixer. Since each packet input to the AC-3 decoder results in 6 output packets (of 256 samples each), this translates to a requirement that four AC-3 buffers be decoded before the mixer starts. The size of this buffer is controlled by a member of the instance setup structure. Zero is a valid size, and the array of buffer pointers is dynamically allocated.

### Delay

In order to satisfy the requirement of adjustable delay for the center and surround channels, the mixer implements a a delay line that can hold a user defined number of packets on the multichannel input. The inner loop of the mixer is implemented with a "preamp" phase followed by a bass redirection phase. In the preamp phase, channel specific gains are applied and samples are selected from the delay line according to the user's specification. The bass redirection phase can then operate on a single output buffer to prepare the data for presentation. The DTV mixer does not implement negative delays. Assuming that 30ms is enough delay, and assuming that each packet is 256 samples at 48000hz (as in Dolby Digital AC-3), a delay line of six packets is enough. The size of this buffer is controlled by a member of the instance setup structure. Zero is a valid size, and the array of buffer pointers is dynamically allocated.

## Bass Redirection

Bass redirection is implemented according the specifications in the Dolby Licensee Implementation Manual, V 2.0 (Dolby LIM 2.0). Seven modes are implemented, including variations on each of the three modes specified by Dolby. These modes have been tested by Dolby and found to be in compliance with the specifications. See figures 2 through 8.

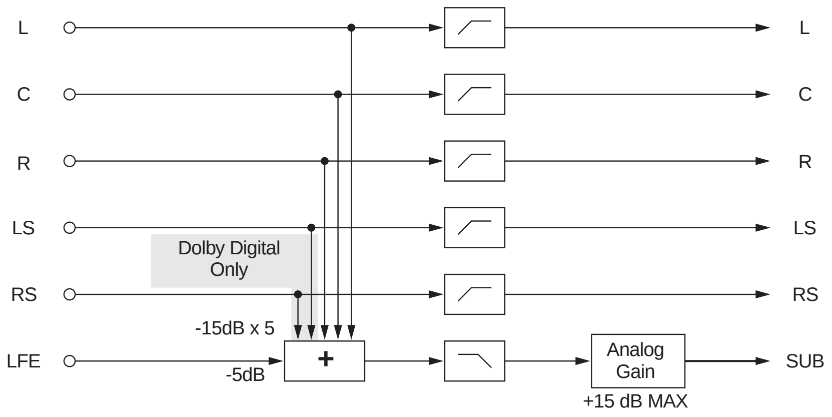


**Figure 7** AMIX_bassRedirHPF_all_a: Dolby bass redirection mode 1: High Pass Filter All

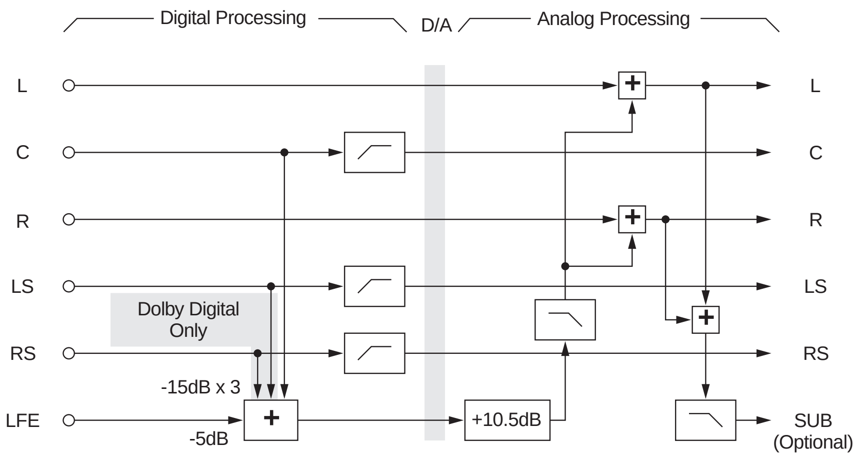


**Figure 8** AMIX_bassRedirFullRangeLR_a: Dolby Bass redirection mode 2 (alternative)

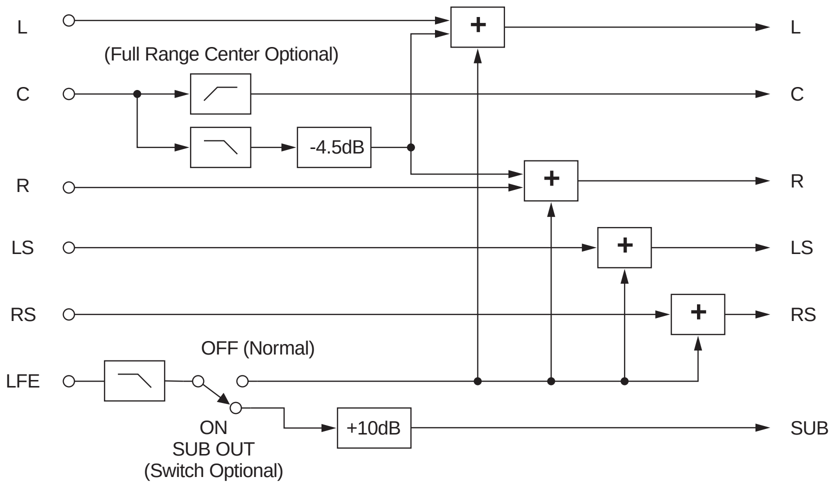


**Figure 9** AMIX_bassRedirFullRangeLRS_s: Dolby Bass redirection mode 3

## DTV Audio Mixer Progress

The DTV Audio mixer does not require a progress function It uses the default progress function to handle changes of format.

## DTV Audio Mixer Errors

Errors can be reported during the mixer's setup phase, or at run time. Errors reported during the setup phase will be noticed as non-zero return values from the API. The definition of these constants is found in tmolAmixDtv.h. In addition, the library used in its debugging mode will use the assert mechanism to flag invalid inputs. These errors are covered along with the descriptions of each function in the API.

The DTV Audio mixer supports the installation of an error callback function. This function is invoked for a set of run time errors, as described below. None of the errors handled by the error callback function are considered fatal. The error function prototype is of the type **tsaErrorFunc_t:**

```
typedef tmLibappErr_t(*tsaErrorFunc_t)(Int instId, UInt32 flags,
ptsaErrorArgs_t args);
typedef struct tsaErrorArgs {
   Int      errorCode;
   Pointer  description;
} tsaErrorArgs_t, *ptsaErrorArgs_t;
```

Handlers should be provided for these possible values of the errorCode:

| | |
|---|---|
| `TMLIBAPP_ERR_UNDERRUN` | The mixer requested data but none was available. Since the DTV mixer blocks indefinitely on the MC input, no underruns will be logged on the MC channel. When an error occurs, the error description is an integer pointer, and description[0] is the ID of the offending channel. |
| `AMIX_ERR_IO_BUFFER_SIZE_MISMATCH` | Triggered when the dataSize of the received packet is not the same as samplesPerPacket (specified in the instance setup) and outChan.bytesPerSample, computed from the initial format. The error description is an integer pointer. |

```
description[0]:   channel ID. 256 for output channel
description[1]:   dataSize of offending packet
description[2]:   expected packet size.
description[3]:   ID of offending packet.
```

## DTV Audio Mixer Configuration

The DTV Audio mixer provides a queue-based configuration function. It can be used to change the volumes of the various channels and the filter cutoff frequency. The queue-based implementation is discussed in some depth in Book 3, *Software Architecture*.

```
tmLibappErr_t tmolAmixDtvInstanceConfig (
   Int instance,
   UInt32 flags,
   ptsaControlArgs_t args
);
```

Values for the command entry in the args structure are defined in tmolAmixDtv.h, and are described in Table 3 on page 244.

A number of commands control volumes. Volume is specified in units of 100th dB. Zero is no gain. The maximum value is 1200, or 12 dB of gain. This is a multiplication by 4 (approximately). No minimum value is specified, but a value of –9600 will attenuate by 96 dB, effectively muting a 16-bit signal. A value of –600 corresponds to a 6 dB attenuation, or a multiplication by 0.5 (approximately).

## DTV Audio Mixer API Data Structures

This section presents the data structures contained in the DTV Audio Mixer library.

| Name | Page |
|---|---|
| tmolAmixDtvCapabilities_t | 235 |
| tmolAmixDtvInstanceSetup_t | 236 |

## tmolAmixDtvCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmolAmixDtvCapabilities_t; *ptmolAmixDtvCapabilities_t;
```

### Fields

| | |
|---|---|
| `defaultCapabilities` | For compliance with the application library architecture, this is a pointer to a structure of the standard type. |

### Description

This structure describes the capabilities and requirements of the DTV Audio mixer module. A user can retrieve the structure's address by calling **tmolAmixDtvGetCapabilities**.

## tmolAmixDtvInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   Int                         masterVolume;
   Int                         multiChannelVolume;
   Int                         auxVolume;
   Int                         headphoneVolume;
   Int                         centerDelay;
   Int                         surroundDelay;
   UInt8                       speakerMode;
   UInt8                       bassRedirMode;
   Int                         crossoverFrequency;
   Int                         trim[NUM_MULTICHANNEL_OUTPUTS];
   Int                         samplesPerPacket;
   UInt8                       delayBufferSize;
   UInt8                       primeBufferSize;
} tmolAmixDtvInstanceSetup_t; *ptmolAmixDtvInstanceSetup_t;
```

### Fields

| | |
|---|---|
| `defaultSetup` | For compliance with TSA, this is a pointer to a structure of the standard type. |
| `masterVolume`, `multiChannelVolume`, `auxVolume`, `headphoneVolume` | Integers specifying volumes in 0.01 dB. The master volume is the output volume. Other volumes control individual channels. Volumes are clipped at +12 dB (1200). More information on volumes is given in the description of control parameters. |
| `centerDelay`, `surroundDelay` | To compensate for the position of speakers in a room, the center and surround channels can be delayed. The delay is specified in microseconds, and it is referred to the sample rate of the output channel. The delay is limited to the length of 8 output packets. Dolby recommends that the user interface limit center channel delay to 5ms and surround delay to 20 ms. |
| `speakerConfig` | Not used. |
| `bassRedirMode` | Per the Dolby spec, several bass redirection modes are supported. They are described in some detail in the section on control parameters. Defaults to zero, or no bass redirection. |
| `crossoverFrequency` | A set of two pole filters is used to implement bass redirection, and the cutoff frequency of these filters can be specified in Hertz. Values between 60 and 150 Hz are recommended. |

| | |
|---|---|
| `trim[6]` | To compensate for speaker and room mismatch, the volume of each speaker can be adjusted separately. This is mapped to balance type controls in the user interface. |
| `samplesPerPacket` | Since all packets must have the same size, it is specified here. The usual value is 256. |
| `delayBufferSize` | The number of input packets held on the multichannel input to implement delay is specified here. The default value is 6 packets, corresponding to 30 ms with 256 samples per packet and a 48,000 Hz sample rate. |
| `primeBufferSize` | It is possible to make the mixer wait for this given number of packets before processing begins. This feature was created to get rid of the bubbles that can exist in the pipeline at startup. |

### Description

The **tmolAmixDtvInstanceSetup_t** structure is used to describe the initial operation of this instance of the mixer.

## DTV Audio Mixer API Functions

This section presents the functions contained in the DTV Audio Mixer library.

## tmolAmixDtvGetCapabilities

```
tmLibappErr_t tmolAmixDtvGetCapabilities(
   ptmolAmixDtvCapabilities_t   *pCap
);
```

### Parameters

| | |
|---|---|
| `pCap` | Pointer to a variable in which to return a pointer to capabilities data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Used to retrieve the capabilities of the DTV Audio mixer. The function pointer that is returned remains valid as long as the mixer is active.

## tmolAmixDtvOpen

```
tmLibappErr_t tmolAmixDtvOpen(
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Pointer (returned) to the instance. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | No more instances of the DTV Audio mixer ar available. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Memory allocation for the default instance variables failed. |

### Description

The open function creates an instance of the DTV Audio mixer and informs the user of its instance. The DTV Audio mixer does support multiple instances, but this feature is untested.

## tmolAmixDtvGetInstanceSetup

```
tmLibappErr_t tmolAmixDtvGetInstanceSetup(
   Int                            instance,
   ptmolAmixDtvInstanceSetup_t   *setup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance, as returned by **tmolAmixDtvOpen**. |
| `setup` | Pointer to a variable in which to return a pointer to setup data. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

A pointer to the current instance setup is retrieved. During a call to **tmolAmixDtvOpen**, this structure is filled with default values to simplify the impending call to **tmolAmixDtv-InstanceSetup**.

## tmolAmixDtvInstanceSetup

```
tmLibappErr_t tmolAmixDtvInstanceSetup(
   Int                          instance,
   ptmolAmixDtvInstanceSetup_t  setup
);
```

### Parameters

| | |
|---|---|
| `instance` | As returned from **tmolAmixDtvOpen**. |
| `setup` | Points to a setup structure as described above. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |
| `TMLIBAPP_ERR_MODULE_IN_USE` | Specified instance does not match current instance. Digitizer supports only one instance. |
| `TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE` | An unsupported data format was requested. |
| `TMLIBAPP_ERR_MEMALLOC_FAILED` | Memory allocation failed. |

Other errors are possibly reported by the device library or board support package.

### Description

The DTV Audio mixer is prepared for operation. Parameters are checked. The mixer is left "stopped." It will become operational on a call to **tmolAmixDtvStart**.

## tmolAmixDtvStart

```
tmLibappErr_t tmolAmixDtvStart(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | As returned from **tmolAmixDtvOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

The mixer represented by the instance is started. An independent task is started to execute the processing described in the section on the mixer's operation.

## tmolAmixDtvStop

```
tmLibappErr_t tmolAmixDtvStop(
   Int   instance,
);
```

### Parameters

| | |
|---|---|
| `instance` | As returned from **tmolAmixDtvOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |
| `TMLIBAPP_ERR_INVALID_INSTANCE` | Can be asserted in debug mode. |

### Description

The mixer represented by the instance is stopped. All packets held by the mixer are returned to their respective queues, and the mixer exits its processing loop in accordance with standard TSSA guidelines.

## tmolAmixDtvInstanceConfig

```
tmLibappErr_t tmolAmixDtvInstanceConfig(
   Int                instance,
   UInt32             flags,
   ptsaControlArgs_t  args
);
```

### Parameters

| | |
|---|---|
| `instance` | As returned from **tmolAmixDtvOpen**. |
| `flags` | not used by tmolAmixDtvInstanceConfig. |
| `args` | Points to a control structure used to modify the operation of the DTV Audio mixer. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

Errors detected by the underlying **tmalAmixDtv** call can be found in the retval member of the control structure.

### Description

While a module is operating, the configuration function can be used to change the operating parameters. The acceptable parameters are described in Table 3, following.

**Table 3** Configuration Parameters

| **Parameter** | **Description** |
|---|---|
| AMIX_CONFIG_MASTER_VOLUME | The master gain is applied to the sum of the multichannel input and the auxiliary input to the mixer. This should be used as a master volume control. The parameter field of the argument structure is a treated as pointer to an integer. |
| AMIX_CONFIG_MULTICHANNEL_VOLUME | The fixed point volume specification passed in the parameter field is applied to the multichannel input to the mixer. It is designed to be used in situations where the more than one audio source is being mixed and displayed through the speaker system. The parameter field of the argument structure is a treated as pointer to an integer. |

**Table 3** Configuration Parameters

| **Parameter** | **Description** |
|---|---|
| AMIX_CONFIG_HEADPHONE_VOLUME | When the two channel input of the mixer is used as a headphone channel, this control affects the volume of the headphone channel. It is disabled when the second mixer output is used for IEC61937 formatted Dolby Digital (AC-3) data. The parameter field of the argument structure is a treated as pointer to an integer. |
| AMIX_CONFIG_AUX_VOLUME | The aux channel is not yet used. It is designed to allow a secondary source, like an audible feedback from the user interface, to be mixed into the master output. The parameter field of the argument structure is a treated as pointer to an integer. |
| AMIX_CONFIG_TRIM | The trim controls are used to control the relative balance of the audio channels. The trim should be controlled during a speaker setup phase of system operation. The parameter field of the argument structure is a treated as pointer to an array of integer six integers in the standard L, R, C, Sub, Lsur, Rsur order. |
| AMIX_CONFIG_CENTER_DELAY | Per the Dolby Digital spec, the center channel of the multichannel input can be delayed by up to 10ms to compensate for the placement of the center channel speaker. The actual delay is specified in microseconds, and it is limited internally to be less than 8 packets in length. The parameter field of the argument structure is a treated as pointer to an integer. |
| AMIX_CONFIG_SURROUND_DELAY | Per the Dolby Digital spec, the surround channels of the multichannel input can be delayed by up to 20ms to compensate for the placement of the surround channel speaker. The actual delay is specified in microseconds, and it is limited internally to be less than 8 packets in length. The parameter field of the argument structure is a treated as pointer to an integer. |
| AMIX_CONFIG_SPEAKER_MODE | This parameter is not used. The mixer always passes six channels on the multichannel input. |

**Table 3** Configuration Parameters

| **Parameter** | **Description** |
|---|---|
| AMIX_CONFIG_BASSREDIR_MODE | The audio mixer supports bass redirection as described in the Dolby Licencee Information Manual. Version 1 of the mixer implements a set of second order crossover filters. The filter Q is 0.707, and the crossover frequency can be controlled with another command. This command allows the user to select one of the values listed in Table 4 on page 247. The parameter field of the argument structure is a treated as pointer to an integer. |
| AMIX_CONFIG_CROSSOVER_FREQUENCY | The frequency of the crossover filter used for bass redirection can be set using this command. The frequency is specified in hertz as an integer. Crossover frequency should be set between 60 and 500 hz. The parameter field of the argument structure is a treated as pointer to an integer. |
| AMIX_CONFIG_GET_LFE_LEVEL | The Dolby LIM suggests that the user interface be able to display the amount of Low Frequency Energy (LFE) currently in the subwoofer channel. The mixer measures and records the peak level of the subwoofer channel in every packet. This command can be used to retrieve the current level. The address of an integer variable to be updated by the mixer is passed as the parameter member of the control structure. LFE level will reach a maximum of 32k in 16 bit systems and 128k in 20 bit systems. The parameter field of the argument structure is a treated as pointer to an integer. |

**Table 4** Mixer values

| Value | Description |
|---|---|
| AMIX_bassRedirNone | No bass redirection is applied. All channels are passed through without modification. |
| AMIX_bassRedirHPF_all_a | As described on p54 of the Dolby LIM, High pass filtering is applied to L, R, C, and surround outputs. All channels are summed to the LFE channel. It is assumed that 15db of gain will be added to the LFE channel in the analog output stage. This is Dolby mode 1. |
| AMIX_bassRedirHPF_all_d | DO NOT USE. Headroom is compromised. As described on p54 of the Dolby LIM, High pass filtering is applied to L, R, C, and surround outputs. All channels are summed to the LFE channel. The required 15db of gain for the LFE channel is added in the digital domain. This is Dolby mode 1. |
| AMIX_bassRedirFullRangeLR_a | The alternative form of Dolby mode 2, as described on p54 of the Dolby LIM. Full range speakers are available for the front L and R channels. The other channels are high pass filtered at the crossover frequency. The low frequency energy (LFE) from the other channels is redirected into the LFE channel. The final summing of the LFE into the L and R channel (with associated gain) is assumed to be implemented in the analog output stage. |
| AMIX_bassRedirFullRangeLR_d | DO NOT USE. Headroom is compromised. Dolby mode 2, as described on p54 of the Dolby LIM. Full range speakers are available for the front L and R channels. The other channels are high pass filtered at the crossover frequency. The low frequency energy (LFE) from the other channels is redirected into the LFE channel. All processing, including the final redirection of the LFE into the L and R channels, is done in the digital domain. |

**Table 4** Mixer values

| **Value** | **Description** |
|---|---|
| AMIX_bassRedirFullRangeLRS_s | Dolby mode 3, as described on p56 of the Dolby LIM: Full range speakers are assumed for L, R, and surround channels. The LFE from the center channel is redirected. A subwoofer is assumed to be present. |
| AMIX_bassRedirFullRangeLRS_ns | Dolby mode 3, as described on p56 of the Dolby LIM: Full range speakers are assumed for L, R, and surround channels. The LFE from the center channel is redirected. Gains are set as if no subwoofer is present, and the subwoofer output is silent. |
| AMIX_bassRedirExtern | As described on p71 of the Dolby LIM: All channels are given full range audio and the all channels are summed into the LFE channel. Bass redirection is assumed to be handled externally. |

Chapter 11

# DTV Audio System (AudSys) API

**Note**
This component library is not included with the basic TriMedia SDE, but is available as a part of other software packages, under a separate licensing agreement. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

## DTV Audio System Overview

The audio system for the TriMedia DTV system meets the requirements of a digital TV system. The requirements are outlined by the Dolby Digital Licensee Information Manual [1]. These requirements include the ability to decode AC-3 and ProLogic encoded signals with six channels of audio output.

TriMedia further extends this specification by adding an extra stereo output that can be used to present digitally encoded AC-3 data, or analog audio.

This most recent release of the audio system includes significant enhancements, including digital tone control and loudness compensation.

The audio system is constructed using the framework of the TriMedia Software Streaming Architecture (TSSA). The system presents a TSSA compatible interface, and it is constructed of several modules, all of which conform to TSSA:

- Audio Renderer
- Dolby Digital AC-3 decoder
- Dolby ProLogic decoder
- Audio Mixer including IEC61937 encoding for SPDIF transmission of Dolby Digital data
- Audio Digitizer
- Noise sequencer

TSSA provides a crucial integrating framework for all of the audio modules. The audio system contains the logic needed to connect these modules and it can modify these connections based on the format of the input stream and the wishes of a user.

The input to the audio system can come from a TSSA compliant source of an AC-3 elementary stream. This might be a transport stream demultiplexer. Or the audio system can be configured to read its input from the audio digitizer that is connected to jacks on the back panel of the system.

The use of the audio system is demonstrated with an example program, exAudSys.

### Statement of Dolby Compliance

In June of 1998, the digital portion of the TriMedia audio system was confirmed to meet all group A specification given by Dolby in reference [1]. The system as it exists today is based on that code, but it will be resubmitted for testing during the summer of 1999.

### New Features in Version 2

The new audio system incorporates the features of the new audio mixer, and expands many of the capabilities already present in the version 1.1. In particular, it allows to control the tone control and the loudness compensation of the mixer. The tone control consists of two boost/cut filters, the bass and the treble filters. The gain/ attenuation of each of these filters can be dynamically controlled by passing its value to the audio system. The bass /treble gain/attenuation can be controlled in the range –12 to +12 dB. The loudness filter's task is to compensate for the subjective sensation of the low and high frequencies loss relative to the mid range of frequencies, at low sound levels. The operation of this filter is controlled by the master volume. Both tone and loudness control filters can be turned on or off.

Auto detection of PCM or IEC61937 format data at the digital input is now provided. The **-mode dauto** command line option to exAudSys enables the auto detection. The audio system waits for the digital bitstream from the digitizer and switches to appropriate playback of the AC-3 or the SPDIF when the bitstream is provided. It also reports the lock loss through the call to the progress function and allows to reconnect the input bitstream from one digital mode to the other. This facility takes some MIPS, and in products it should be augmented (or replaced) by the facility to check the state of the non-PCM C bit in the SPDIF channel status structure.

The “headphone” output of the audio system can now be switched between PCM coded (headphone) output, or IEC61937 coded (SPDIF) output. The operator of exAudSys can dynamically switch from one form to another by issuing the **hpm** command.

The ProLogic decoder is now called as a function from the mixer. In the past, it was operated as a task. This change allows us to reduce latency from input to output when decoding ProLogic, as would be required with an analog TV signal.

In addition to the ProLogic algorithm for converting stereo input to 5.1, an “echo” mode has been added. This simply copies the front channels to the rear channels with a delay of a few milliseconds.

The noise sequencer was extended to call a progress function when it switches from one channel to another in its auto-rotate mode. Also, the time spent in each channel can now be controlled with a configuration command.

The audio system 2.0 can also use the new SPDIF output of the TM1300 and TM2. These new microprocessors allow the user to pass a selected channel pair of the PCM playback to the SPDIF output. This feature is added to enable testing with the digital output provided on TM2700. The new arendSpdif renderer controls appropriate formatting and transmission of the SPDIF stream to the output. To use this feature an **-tm2spdif** option has to be used when executing the exAudSys. The SPDIF channel pair indicator (a number between 0 and 3) has to be used with this option to start the desired playback. The channel pair parameter can be then dynamically changed (see the “spdif” of the exAudSys).

## Programmers Interface

The programmers interface to the audio system is like that of any TSSA component.

- tsaAudSysOpen
- tsaAudSysClose
- tsaAudSysGetInstanceSetup
- tsaAudSysInstanceSetup
- tsaAudSysInstanceConfig
- tsaAudSysStart
- tsaAudSysStop
- ErrorCallbackFunction
- ProgressCallbackFunction

## Required Board Support

Proper operation of the audio system requires that certain features be implemented in the board support package. Among these are:

- Audio out clock must be provided by TriMedia's AO DDS.
- Audio in clock should be derived from AI DDS if an analog source is used, but it will be recovered from the SPDIF source if an SPDIF source is used.
- For digital input to be used, board support must be provided for digital audio input. This should include the control interrupt that is generated when the state of the digital input changes. This can be tested using the program exolCopy Audio, as it uses mechanisms similar to those of AudSys.

## Related Documents

[1] Dolby Digital Licensee Information Manual version 2.0. April, 1997.

Details of the APIs of the underlying audio system components are available in the TriMedia Developer's kit documentation:

- Audio Renderer (ArendAO)
- Audio Digitizer (AdigAI)
- Dolby Digital (AdecAc3)
- Dolby ProLogic (AdecPl)
- DTV Audio Mixer (AmixDtv)

## DTV Audio System Inputs and Outputs

The audio system makes most of its connections to the outside world through the audio renderer and digitizer. To the hosting program, it has one input port for AC-3 encoded elementary streams. It also has a control port. Figure 10 gives a simplified idea of the inner structure.

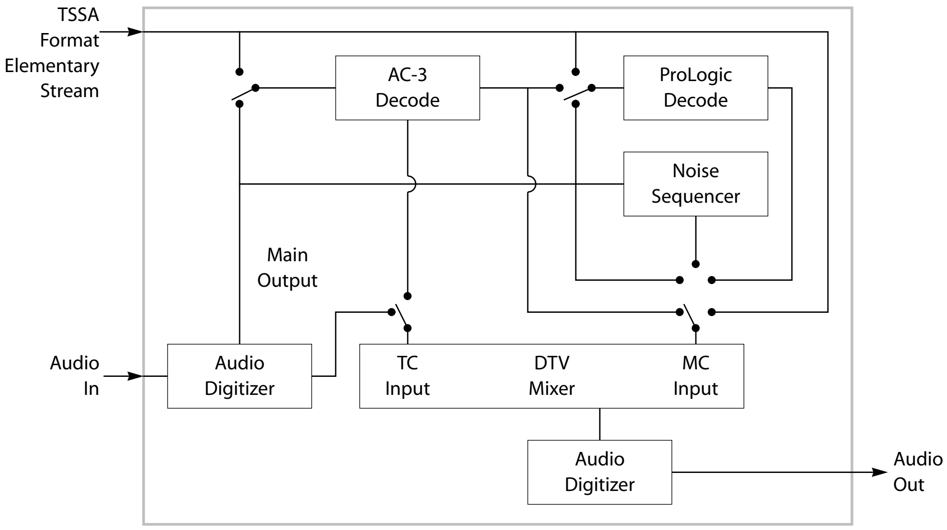


**Figure 10** DTV Audio System Inputs and Outputs

The renderer and digitizer are assumed to be capable of analog or digital input and output. Through the DTV Mixer, the renderer should support operation at 48 kHz with 8 channels of 20-bit data. The digitizer supports only 16-bit stereo data, but the input should be selectable between an analog input and an SPDIF digital input. The selection is made with an appropriate call to the board support package.

## DTV Audio System Errors

Users of the audio system can install an error handling function. The standard TSSA prototype is used:

```
typedef struct tsaErrorArgs {
   Int      errorCode;
   Pointer  description;
} tsaErrorArgs_t, *ptsaErrorArgs_t;

tmLibappErr_t tsaErrorFunc( Int instId, UInt32 flags,
                            ptsaErrorArgs_t args);
```

The function is called when:

- renderer errors occur
- mixer errors occur
- digitizer errors occur
- errors are encountered in the AC-3 data stream

The error function should switch on the error code specified in the arguments to the error function. Since each of the errors reported by the subsidiary components is unique, it is simply passed up to the user of the audio system. An example of an appropriate error handler can be found in exAudSys.c

The error handler should return zero if the error has been completely handled. In the case of the renderer or digitizer, the error function may be called from within an interrupt handler, so care should be taken not to create a deadlock with a call to **printf**.

## DTV Audio System Progress

The audio system progress function is called in at least three cases. An example of an appropriate progress function is found in exAudSys.c. The progress function can report when the first AC-3 sync word is found in a new stream. It is also called when the AC-3 bitstream information says that the stream is ProLogic encoded. And the progress function reports on the state of the audio video (AV) synchronization mechanism, as maintained by the audio renderer. This latter case gives the user the chance to customize the AV sync algorithm.

## DTV Audio System Configuration

The audio system implements the TSA standard of a single function to configure the audio system. This function uses a switch statement to handle the various commands. Refer to **tsaAudSysInstanceConfig** for more information about the function and its commands.

## DTV Audio System Operation

The system can be operated in a number of different modes. These modes are specified using the contents of the setup structure. The mode of operation can also be changed on the fly using the configuration function.

The following sections describe two representative modes.

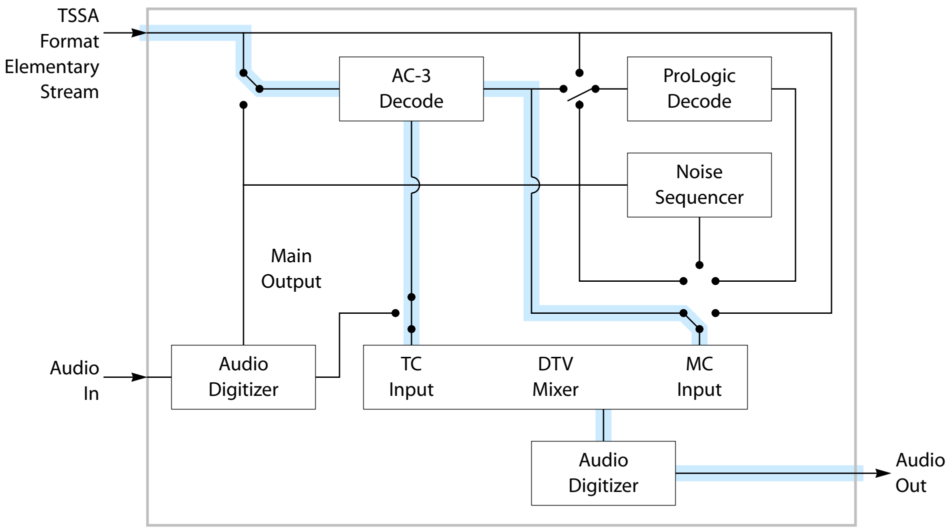


**Figure 11** DTV Audio System Decoding Dolby Digital AC-3 Elementary Stream

Figure 11 illustrates the basic DTV decoding mode. The AC-3 decoder is active, and the mixer is interleaving the six channels of PCM data with the two channels of IEC61937 format data for output on the SPDIF connector. This mode is selected by setting:

```
AudSysSetup.inputSource = audSysInputElementaryStream
```

When the standard start sequence is invoked, the action is as described below:

```
tsaAudSysOpen(&audSysInstance);
```

The capabilities of all the internal components are retrieved, and all internal components are opened. The internal instance setup structure is allocated and filled with default values. The values that are not zero are initialized to:

```
AudSysSetup.speakerConfig      = audSysSpeakerConfig_3_2;
AudSysSetup.bassRedirMode      = AMIX_bassRedirNone;
AudSysSetup.subwoofOn          = True;
AudSysSetup.proLogicEnable     = audSysProLogicOff;
AudSysSetup.compressionMode    = audSysCompStandard;
AudSysSetup.effectsMode        = audSysEffectsNone;
AudSysSetup.noiseSequencerOn   = False;
AudSysSetup.plAutoBalanceOn    = True;
AudSysSetup.plWideSurroundOn   = False;
AudSysSetup.numMixPackets      = 26;
AudSysSetup.numArPackets       = 6;
AudSysSetup.numAiPackets       = 26;
AudSysSetup.numMixPrimeBuffers = 19;  /* three AC-3 blocks + 1 packet */
AudSysSetup.inputSource        = audSysInputAnalogIn;
AudSysSetup.audioPriorityBase  = 100;
AudSysSetup.crossoverFrequency = 120;
AudSysSetup.centerDelay        = 0;       /* microsec */
AudSysSetup.surroundDelay      = 5000;    /* (5 ms in microsec) */
AudSysSetup.dynRngScaleHi       = -1.0;    /* if neg, use defaults */
AudSysSetup.dynRngScaleLow      = -1.0;    /* if neg, use defaults */
AudSysSetup.sampleRate          = 48000.0;
AudSysSetup.headphoneMode       = audSysHeadphone61937;
AudSysSetup.inversionRequired   = False;
AudSysSetup.syncThreshold       = 3000;
AudSysSetup.framesToMute        = 1;
AudSysSetup.syncMode            = AR_SyncMode_skip;
```

When **tsaAudSysInstanceSetup** is then called, the internally held setup structure is updated to match that specified. If this is the first time that **InstanceSetup** is called, the InOut descriptors that are used to connect the various internal modules are created. The queues are created and the packets that circulate in the queues are malloc'ed. In the default configuration, about 600K of buffer memory is allocated. By avoiding the use of the prime buffer, that number can be lowered to near 400K. Enough buffers are allocated during **InstanceSetup** to cover the worst case mode of operation.

It is in **tsaAudSysStart** that the connections between the various modules are made and the internal modules are started. **tsaAudSysStart** can be called multiple times. Except for the renderer, which always runs, all internal components are stopped momentarily before restarting them with the correct connections. Memory is not reallocated.

A more complex configuration is shown in Figure 12. Here, two more components are running under the control of the audio system.

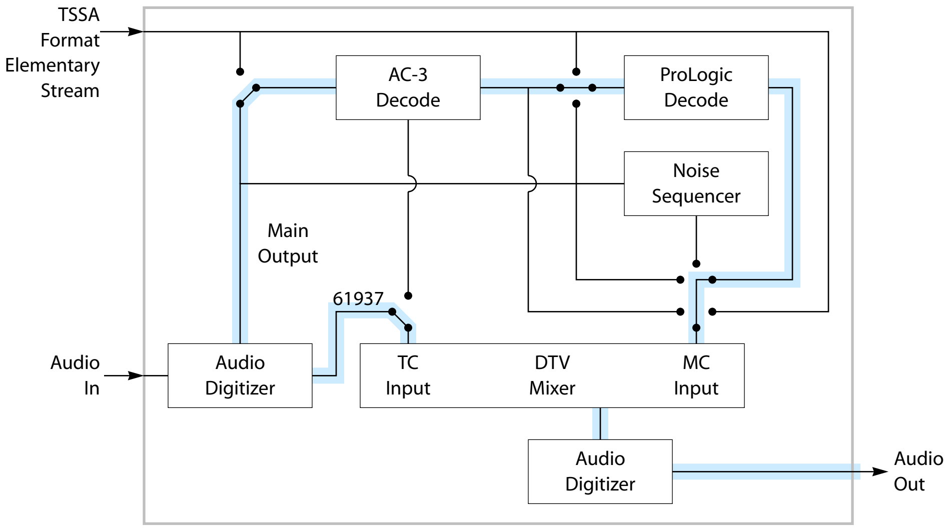


**Figure 12** DTV Audio System Decoding ProLogic-Encoded AC-3 Stream from SP DIF Input

Each of the blocks in these pictures represents a separate thread of execution. While the renderer and the digitizer are interrupt driven, the rest of the components are tasks. In addition, the audio system creates a task that is used to check for changes that might necessitate automatic configuration. Examples of this are changes in sample rate detected in an AC-3 stream. Or a stream might change format from 3_2 into ProLogic encoded stereo. When ProLogic is set up in automatic mode, the audio system task will cause the ProLogic decoder to be started.

### Resource Usage

The numbers given below are ballpark figures measured under typical conditions.

#### Text Memory

The combined text, data and bss segments of the audio system libraries require about 260k. This includes about 30K of shared TSSA default functions.

#### Data Memory

The audio system allocates at least 400K for audio buffers. This is under the control of the user via the buffer count settings. Typical operation requires 600K. This memory is allocated using the **malloc** or **_cache_malloc**. Most of it is allocated by the **tsaDefaultI-nOutDescriptorCreate** function calls.

#### MIPS

The audio system example (exAudSys.out) can report MIPS usage dynamically. Typical six-channel AC-3 bitstreams require about 25 MIPS. ProLogic decode (when enabled) requires about 7 MIPS. The audio mixer requires about between 10 and 20 MIPS in this release, but this will be reduced in future releases.

## DTV Audio System API Data Structures

This section presents the DTV Audio System API data structures.

| **Name** | **Page** |
|---|---|
| Audio System Constants | 258 |
| tsaAudSysStatus_t | 260 |
| tsaAudSysSetup_t | 262 |

## Audio System Constants

These constants are defined in tsaAudSys.h. They are designed to communicate configuration specifications to the audio system.

Values for **inputSource**:

| | | |
|---|---|---|
| audSysInputElementaryStream | 0 | |
| audSysInputAnalogIn | 1 | |
| audSysInputDigitalIn | 2 | |

Values for **decodeMode**:

| | | |
|---|---|---|
| audSysDecodeModeAutoDetect | 0 | Not fully implemented |
| audSysDecodeModeAC3 | 1 | |
| audSysDecodeModePCM | 2 | |

Speaker Configurations:

| | | |
|---|---|---|
| audSysSpeakerConfig_PL | 0 | ProLogic encoded stereo |
| audSysSpeakerConfig_1_0 | 1 | Center channel mono |
| audSysSpeakerConfig_2_0 | 2 | Normal stereo |
| audSysSpeakerConfig_3_0 | 3 | Stereo plus center speaker |
| audSysSpeakerConfig_2_1 | 4 | Two front speakers, one surround speaker |
| audSysSpeakerConfig_3_1 | 5 | Three front speakers, one suround speaker |
| audSysSpeakerConfig_2_2 | 6 | Two front speakers, two suround speakers |
| audSysSpeakerConfig_3_2 | 7 | Three front speakers, two suround speakers |

Values for **processMode**:

| | | |
|---|---|---|
| audSysEffectsNone | 0 | Default |
| audSysEffectsTHX | 1 | Not implemented |
| audSysEffectsTheater | 2 | Not implemented |
| audSysEffectsHall | 3 | Not implemented |
| audSysEffectsStadium | 4 | Not implemented |

Values for **compressionMode**. See Dolby LIM, table 7.11 [1]:

| | | |
|---|---|---|
| audSysCompDirect | 0 | No compression applied, requires analog dialog normalization. |
| audSysCompMaximum | 1 | Line mode, No compression applied |
| audSysCompStandard | 2 | Line mode, full compression (default) |
| audSysCompLateNight | 3 | RF mode, fully compressed, with HF rolloff, and 11 dB gain. |

Values for **proLogicEnable**:

| | | |
|---|---|---|
| audSysProLogicOff | 0 | Always off. |
| audSysProLogicOn | 1 | On whenever input is stereo. |
| audSysProLogicAuto | 2 | Enables PL if AC-3 stream is labeled as PL |

Values for **headphoneMode**:

| | | |
|---|---|---|
| audSysHeadphoneOff | 0 | |
| audSysHeadphonePCM | 1 | (not implemented) |
| audSysHeadphone61937 | 2 | (default) |

These values are used with the **audSysConfig** function to set the decode mode.

| | | |
|---|---|---|
| audSysAutoDetectOn | 0 | (not implemented) |
| audSysAutoDetectForceAC3 | 1 | |
| audSysAutoDetectForcePCM | 2 | |

## tsaAudSysStatus_t

```
typedef struct {
   Int32    LFE_level;
   Bool     audioInputIsPCM;
   UInt8    programSpeakerConfig;
   Bool     programSubwoofIsOn;
   Bool     programProLogicEncoded;
   UInt16   programDataRate;
   Int      programSampleRate;
   UInt8    programBitStreamIdentification;
   UInt8    programBitStreamMode;
   UInt8    programCenterMixLevel;
   UInt8    programSurMixLevel;
   UInt8    programDialogueNormalization;
   Bool     programLanguageCodeExists;
   UInt8    programLanguageCode;
   Bool     programAudioProductionInfoExists;
   UInt8    programMixingLevel;
   UInt8    programRoomType;
   Bool     programCopyrightProtected;
   Bool     programOriginalBitstream;
   UInt8    numAc3Errs;
   UInt8    numArendErrs;
   UInt8    numMixerErrs;
   UInt8    numDigitizerErrs;
} tsaAudSysStatus_t, *ptsaAudSysStatus_t;
```

### Fields

| | |
|---|---|
| `LFE_level` | Reflects the peak level of the signal present in the LFE channel. The peak level is updated every five milliseconds. |
| `audioInputIsPCM` | False if the AC-3 decoder is running and it has detected a valid AC-3 sync word in the bitstream. |
| `programSpeakerConfig` | Reflects the format of the AC-3 audio stream as encoded. The value will be one of the **audSys-SpeakerConfig** constants defined above. |
| `programSubwoofIsOn` | True if the AC-3 program stream includes an LFE channel. |
| `programProLogicEncoded` | True if the AC-3 program is encoded as stereo with ProLogic encoding. |
| `programDataRate` | The data rate of the AC-3 stream, in kilobits per second. |
| `programSampleRate` | The nominal sample rate of the incoming AC-3 stream, in Hertz. |

| | |
|---|---|
| `numAc3Errs` | Records the number of errors reported by the AC-3 decoder since last status check. |
| `numArendErrs` | Records the number of errors reported by the audio renderer since last status check. |
| `numMixerErrs` | Records the number of errors reported by the audio mixer since last status check. |
| `numDigitizerErrs` | Records the number of errors reported by the audio digitizer since last status check. |

## Description

This structure is used by the application to find out about the status of the audio system. In particular, it retrieves information about an AC-3 encoded input stream. The command **AUDSYS_COMMAND_GET_STATUS** can be used to access an updated copy of this structure.

## tsaAudSysSetup_t

```
typedef struct {
   UInt8                    inputSource;
   UInt8                    speakerConfig;
   UInt8                    bassRedirMode;
   Bool                     subwoofOn;
   UInt8                    proLogicEnable;
   UInt8                    compressionMode;
   UInt8                    effectsMode;
   Bool                     mixerBypass;
   UInt8                    decodeMode;
   Bool                     noiseSequencerOn;
   Bool                     noiseCenterExtendedMode;
   Float                    noiseDwellTime;
   Bool                     plAutoBalanceOn;
   Bool                     plWideSurroundOn;
   UInt8                    ac3KaraokeMode;
   ptsaInOutDescriptor_t    esIod;
   ptsaClockHandle_t        PCRClock;
   UInt32                   audioPriorityBase;
   UInt16                   crossoverFrequency;
   Int16                    masterVolume;
   Int16                    mainVolume;
   Int16                    headphoneVolume;
   Int16                    auxVolume;
   Int32                    trim[6];
   UInt16                   centerDelay;
   UInt16                   surroundDelay;
   UInt16                   echoDelay;
   Float                    dynRngScaleHi;
   Float                    dynRngScaleLow;
   Float                    sampleRate;
   tsaErrorFunc_t           errorFunc;
   tsaProgressFunc_t        progressFunc;
   UInt8                    numMixPackets;
   UInt8                    numArPackets;
   UInt8                    numAiPackets;
   UInt8                    numMixPrimeBuffers;
   UInt8                    headphoneMode;
   UInt8                    inversionRequired;
   UInt16                   framesToMute;
   UInt16                   syncThreshold;
   arSyncMode_t             syncMode;
   UInt16                   progressReportFlags;
   UInt16                   errorReportFlags;
   UInt8                    toneGenOn;
   UInt8                    toneGenChannel;
   Float                    toneGenFrequency;
   Bool                     toneControlEnable;
   loudnessMode_t          loudnessMode;
   Int                     alingmentVolume;
   Bool                    useSpdifArend;
   Int                     spdifChanPair;
   Int                     packetBase;
} tsaAudSysSetup_t, *ptsaAudSysSetup_t;
```

## Fields

| | |
|---|---|
| `inputSource` | Select an input source for the audio system. Use one of the **audSysInputSource** constants as defined in tsaAudSys.h, and described above. |
| `speakerConfig` | Select output speaker configuration. Use one of the **audSysSpeakerConfig** constants as defined in tsaAudSys.h, and described above. |
| `bassRedirMode` | Bass redirection mode. As defined in tsaAmixDtv.h, and described in the section on **tsaAudSysConfig**. |
| `subwoofOn` | Enable or disable generation of LFE channel by AC-3 decoder. |
| `proLogicEnable` | Disable (**0**), enable (**1**), or automate (**2**) ProLogic decoding. ProLogic decoding will only be applied to stereo input streams. The automatic mode uses the indication given in an AC-3 stream to control the decoder. |
| `compressionMode` | Dynamic range compression modes: See description under **tsaAudSysInstanceConfig**. |
| `effectsMode` | Effects mode. Reserved. Set to zero. |
| `mixerBypass` | If true, the mixer is bypassed and AC-3 is run in stereo, 16-bit mode and connected directly to the audio renderer. |
| `decodeMode` | Controls how the decoder handles inputs.<br>**audSysDecodeModeAc3** will force AC-3 decoding.<br>**audSysDecodeModePCM** will force PCM playback.<br>**audSysDecodeModeAutoDetect** is not yet implemented. |
| `noiseSequencerOn` | True to use noise sequencer as multi-channel audio source. Used to setup Dolby Digital systems. |
| `noiseCenterExtendedMode` | If true, center channel gets 50% more dwell time. |
| `noiseDwellTime` | Dwell time in seconds for the Noise Sequencer. |
| `plAutoBalanceOn` | True to enable ProLogic autobalance feature. Default is True. |

| | |
|---|---|
| `plWideSurroundOn` | True to enable ProLogic wide surround feature. Default is False. |
| `ac3KaraokeMode` | Sets mode of operation of AC-3 when Karaoke features are in use. See AC-3 documentation. |
| `esIod` | Pointer to IO Descriptor for Elementary Stream input. |
| `PCRClock` | The application supplies this clock to be used as the reference for AV sync. The expected frequency is 90 kHz. A Null entry disables AV sync. |
| `audioPriorityBase` | Audio task priorities are given as offsets from this. Valid range is 10–230. AC-3 is assigned this priority plus 3. ProLogic gets this plus 2. The mixer gets priorityBase plus one. |

The following eight parameters are passed to the audio mixer (AmixDtv). Refer to that documentation for more information.

| | |
|---|---|
| `crossoverFrequency` | Crossover frequency for bass redirection, in hertz. Values are expected to range from 60 to 150 Hz (currently limited to the range from 80 to 120 Hz). |
| `masterVolume` | Master output volume, in 0.01 dB. |
| `mainVolume` | Main output volume, in 0.01 dB. |
| `headphoneVolume` | Headphone output volume, in 0.01 dB. |
| `auxVolume` | 0.01 dB. |
| `trim` | Per channel volume trim, in 0.01 dB. |
| `centerDelay` | Given in microseconds. |
| `surroundDelay` | Given in microseconds |
| `echoDelay` | Given in microseconds |
| `dynRngScaleHi` | Dynamic range scale high. |
| `dynRngScaleLow` | Dynamic range scale low. Dynamic range controls are used by AC-3. Valid range is 0 to 1.0, with negative values meaning “use defaults.” |
| `sampleRate` | Sample rate, in Hertz. This is normally determined by the header of the AC-3 bitstream. |
| `errorFunc` | Error callback function pointer. |
| `progressFunc` | Progress callback function pointer. |
| `numMixPackets` | Number of packets created in the queues at the input of the mixer. |
| `numArPackets` | Number of packets created in the queue connecting the mixer and the audio renderer. |
| `numAiPackets` | Number of packets created in the queues at the output of the audio digitizer. Should be the same as **numMixPackets**. |

| | |
|---|---|
| `numMixPrimeBuffers` | The mixer can be configured to wait until this many packets have been delivered by the AC-3 decoder before starting to present the audio. This is only done when the input is from the audio digitizer. Setting this to 19 packets allows AC-3 decode to start without glitches. It is OK to set this to zero. |
| `headphoneMode` | **audSysHeadphoneOff** causes the headphone output to be zero. **audSysHeadphonePCM** causes the headphone output to be a stereo mix of the main output. **audSysHeadphone61937** causes the SPDIF output to be coded AC-3 data, as described in IEC standard 61937. |
| `inversionRequired` | Since some D/A converters have been found to invert the polarity of the analog output, this switch was added to correct for the inversion. |
| `framesToMute` | When the renderer encounters a discontinuity in the AV sync algorithm, the output is muted. When the cause of the mute is removed, the mute remains active for this many audio interrupts. This mechanism allows users to avoid the choppy mute-unmute-mute-unmute sequence that can be noticed when changing bit streams. |
| `syncThreshold` | When the audio renderer detects that we are “far” out of lock, a more aggressive sync algorithm is enabled. This parameter, given in clock ticks, determines that transition. A normally reasonable value is 3000, which translates to about one frame at a 90 kHz clock. |
| `syncMode` | Determines the behavior of the portion of the AV sync algorithm. The legal modes are described in the audio renderer’s documentation. **AR_Sync_None** is appropriate if you are not dealing with AV sync. **AR_Sync_skip** is appropriate if you are syncing the audio to a reference clock using time stamps. |
| `progressReportFlags` | Not currently used. All appropriate progress reports are always enabled. |
| `errorReportFlags` | Used to enable or disable error reporting in the various subsidiary components of the audio renderer. |
| `toneGenOn` | Non-zero if the user wishes the mixer to generate a test tone rather than output the mixed version of the input. Note that an input source is required for the tone generator to work. |

| | |
|---|---|
| `toneGenChannel` | A channel mask to control the tone generator. The lowest bit is for the left channel. The channel mapping is defined in tmalAmixDtv.h. |
| `toneGenFrequency` | The test tone can be any frequency in the audio range. If it is below 20 Hz, a noise source is enabled as the test tone. |
| `toneControlEnable` | Enables/disables the tone control of the mixer. The amount of the bass/treble cut/boost can be controlled by issuing appropriate commands to the audio system as described in the Table 5.1 (Configuration Commands). |
| `loudnessMode` | Enables/disables the 'loudness' filter of the mixer. This filter has very little effect on loud sounds (close to 0 dBFS). For the soft sounds, the filter raises the low and high frequencies trying to compensate for the subjective feeling of week bass/treble of such soft sounds. loudnessMode can currently have values of **AMIX_LOUDNESS_STATIC** (**ON**) or **AMIX_LOUDNESS_OFF** (default). |
| `alignmentVolume` | The master and main volume controls can only attenuate sounds (their values can be only expressed as negative dBFS). Often it is important to raise the volume of the soft sounds, however. This can be accomplished by setting/controlling the alignmentVolume, whose values can go positive. The alignmentVolume is set in 0.01 dB as any other volume control.<br><br>The alignment volume is designed to be used by system designers to align their audio channel while tuning the product. The alignment volume changes the position of the zero point. When the alignment volume is zero dB or lower, the system will never clip. As you raise the alignment gain to a maximum volume of +18 dB, it becomes possible to saturate a full scale sound when trim and tone controls are also maximized.<br><br>Default: 0. |
| `useSpdifArend` | Instead of using the AO audio renderer (default), force the output to the SPDIF renderer that is present on TM1300 and TM2. Designed for use in tests, not product.<br><br>Default: False. |

| | |
|---|---|
| `spdifChanPair` | The selector of the SPDIF channel pair to be played by the SPDIF renderer.<br>0: Left and right channels.<br>1: Center and subwoofer.<br>2: Left and right surround channels.<br>3: Headphone channels.<br>Default: 0 |
| `packetBase` | When debugging, it can be useful to monitor the packet ID that is in the header of each packet. This setup entry allows the user to vary the base of the audio packets so that it might not conflict with another component.<br>Default: 0. |

## Description

This structure is used to initialize the audio system.

The call to **tsaAudSysGetInstanceSetup** returns a pointer to the internal copy of this structure. Many of the values in this structure can also be changed using the **tsaAudSys-InstanceConfig** function. More data about the controls may be found in that section.

## DTV Audio System API Functions

This section presents DTV Audio System API functions.

## tsaAudSysOpen

```
tmLibappErr_t tsaAudSysOpen(
   Int   *instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value is determined by this function call. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Open all audio system components.

## tsaAudSysClose

```
tmLibappErr_t tsaAudSysClose(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value as returned by **tsaAudSysOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Close all audio system components.

## tsaAudSysInstanceSetup

```
tmLibappErr_t tsaAudSysInstanceSetup(
   Int                 instance
   tsaAudSysSetup_t   *setup
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value as returned by **tsaAudSysOpen**. |
| `setup` | Pointer to a structure containing information necessary to start the audio system. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Called once, before starting the audio system. Specify queues connecting AC-3 elementary stream source. Specify initial mode, and volumes. The first time this is called, a significant amount of setup is performed. All of the InOutDescriptors that are used internally are allocated.

## tsaAudSysInstanceConfig

```
tmLibappErr_t tsaAudSysInstanceConfig(
   Int                 instance,
   UInt32              flags
   tmalControlArgs_t   *config
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value as returned by **tsaAudSysOpen**. |
| `flags` | Always pass **tsaControlWait**. |
| `config` | Pointer to a structure specifying how to change the configuration of the operating audio system. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. The actual return value from the target component will be found in the **retval** field of the config structure. |

### Description

Use to change the configuration of the operating audio system. Accepted commands are listed as follows:

`AUDSYS_COMMAND_GET_STATUS`

The parameter field of the control structure is filled in with the address of an updated **AudSysStatus** structure. This address remains valid as long as the audio system is active.

`AUDSYS_COMMAND_SET_INPUT`

The input of the audio system can be set as

**audSysInputElementaryStream**
**audSysInputAnalogIn**
**audSysInputDigitalIn**

These values are cast and passed directly in the parameter field of the config structure.

`AUDSYS_COMMAND_GET_INPUT`

The currently selected input (as listed above) is returned in the parameter field of the control structure.

`AUDSYS_COMMAND_SET_SPEAKER_CONFIG`

The Dolby standard speaker configurations are accepted. Symbolic constants representing these are defined in tsaAudSys.h, and are described above. These values are cast and passed directly in the parameter field of the config structure. This value is passed to the currently active decoder, whether AC-3, ProLogic, or the noise sequencer. It does not affect the operation of the mixer.

`AUDSYS_COMMAND_GET_SPEAKER_CONFIG`

The currently selected speaker configuration (see **AUDSYS_COMMAND_SET_SPEAKER_CONFIG** above) is returned in the parameter field of the control structure.

`AUDSYS_COMMAND_SET_SUBWOOFER`

`AUDSYS_COMMAND_GET_SUBWOOFER`

Set subwoofer ON (1) or OFF (0).

`AUDSYS_COMMAND_SET_MUTE`

`AUDSYS_COMMAND_GET_MUTE` Mute the output of the audio system at the renderer. Data continues to flow.

`AUDSYS_COMMAND_SET_COMPRESSION_MODE`

`AUDSYS_COMMAND_GET_COMPRESSION_MODE`

Control the dynamic range compression implemented as part of the Dolby Digital AC-3 decoder. Legal values are defined in tsaAudSys.h:

**audSysCompDirect**
No compression is applied

**audSysCompMaximum**
Compression controlled by dynamic range scale parameters.

**audSysCompStandard**
Compression appropriate for normal TV listening

**audSysCompLateNight**
Full compression suitable for late night TV viewing

`AUDSYS_COMMAND_SET_DYNRANGESCALE`

`AUDSYS_COMMAND_GET_DYNRANGESCALE`

Control the two parameters that allow customization of dynamic range scaling. These are the “low cut” and “high boost” parameters exported by the Dolby Digital AC-3 decoder. Higher values will yield more dynamic range compression at the low and high end of the spectrum. Maximum value is 1.0. Minimum is 0.0. The parameter field of the control structure holds the address of an array of two floating point values.

`AUDSYS_COMMAND_SET_EFFECTS_MODE`

`AUDSYS_COMMAND_GET_EFFECTS_MODE`

Effects modes are not implemented.

`AUDSYS_COMMAND_SET_PROLOGIC_MODE`

`AUDSYS_COMMAND_GET_PROLOGIC_MODE`

**audSysProLogicOff**, **audSysProLogicOn**
Apply ProLogic decode to any two channel stream

**audSysProLogicAuto**
Apply ProLogic decode only to the output of an AC-3 decode that is labeled as ProLogic encoded.

These values are cast and passed directly in the parameter field of the config structure.

`AUDSYS_COMMAND_SET_PL_AUTOBALANCE`

`AUDSYS_COMMAND_GET_PL_AUTOBALANCE`

A control for the ProLogic Decoder. The autobalance feature can be disabled. It defaults to enabled.

`AUDSYS_COMMAND_SET_PL_WIDE_SURROUND`

`AUDSYS_COMMAND_GET_PL_WIDE_SURROUND`

A control for the ProLogic Decoder. The surround channels are normally band limited between 100 Hz and 7 kHz. Enabling wide surround will disable the bandlimiting filters.

`AUDSYS_COMMAND_SET_AC3_KARAOKE_MODE`

`AUDSYS_COMMAND_GET_AC3_KARAOKE_MODE`

Control for Dolby Digital AC-3 decoder

`AUDSYS_COMMAND_SET_HEADPHONE_MODE`

`AUDSYS_COMMAND_GET_HEADPHONE_MODE`

When an 8-channel renderer is used, and the source is AC-3 data, the last channel pair can either be IEC61937 encoded AC-3 data or it can be a PCM downmix of the decoded AC-3 stream. In that case, three legal values are recognized as the headphone mode:

**audSysaHeadphoneOff**
**audSysHeadphone61937**
**audSysHeadphonePCM**

The requested value is cast as a Pointer and passed directly as the parameter of the config structure.

`AUDSYS_COMMAND_SET_AUTODETECT`

`AUDSYS_COMMAND_GET_AUTODETECT`

When autodetect is turned on, the system will attempt to determine automatically whether the input stream is PCM format or AC-3. This is done by checking the “non-PCM audio” bit in the S/P DIF subcode on digital audio in. Legal values are:

**audSysAutoDetectOn**
**audSysAutoDetectForceAC3**
**audSysAutoDetectForcePCM**

The requested value is cast as a Pointer and passed directly as the parameter field of the config structure. This control is expected to be used with digital audio input. In other cases, it should be set to force the correct control mode. This control is not fully implemented. Board support may be missing to read the SPDIF subcode. A more sophisticated test for AC-3 format data would look into the data stream. This is not enabled.

A number of commands control volumes. Volume is specified in units of 0.01 dB. Zero is no gain. The maximum value is 1200, or 12 dB of gain. This is a multiplication by 4 (approximately). No minimum value is specified, but a value of –9600 will attenuate by 96 dB, effectively muting a 16-bit signal. –600 corresponds to a 6 dB attenuation, or a multiplication by 0.5 (approximately).

`AUDSYS_COMMAND_SET_MASTER_VOLUME`

`AUDSYS_COMMAND_GET_MASTER_VOLUME`

The master gain is applied to the sum of the multichannel input and the auxiliary input to the mixer. This should be used as a master volume control. This version of the audio system does not give access to the aux input of the mixer. The integer volume is cast to a Pointer and passed directly as the parameter field of the config structure.

`AUDSYS_COMMAND_SET_MAIN_VOLUME`

`AUDSYS_COMMAND_GET_MAIN_VOLUME`

The main volume is applied to the multichannel input to the mixer. It is designed to be used in situations where the more than one audio source is being mixed and displayed through the speaker system. The integer volume is cast to a Pointer and passed directly as the parameter of the config structure.

`AUDSYS_COMMAND_SET_HEADPHONE_VOLUME`

`AUDSYS_COMMAND_GET_HEADPHONE_VOLUME`

When the two channel input of the mixer is used as a headphone channel, this control affects the volume of the headphone channel. It is disabled when the second mixer output is used for IEC61937 formatted Dolby Digital (AC-3) data. The integer volume is cast to a Pointer and passed directly as the parameter field of the config structure.

`AUDSYS_COMMAND_SET_AUX_VOLUME`

`AUDSYS_COMMAND_GET_AUX_VOLUME`

The aux channel is not yet used. It is designed to allow a secondary source, like an audible feedback from the user interface, to be mixed into the master output. The integer volume is cast to a Pointer and passed directly as the parameter field of the config structure.

`AUDSYS_COMMAND_SET_ALIGNMENT_VOLUME`

`AUDSYS_COMMAND_GET_ALIGNMENT_VOLUME`

The volume of the master channel can only have negative values up to 0 dB. Sometimes it is important to boost low level sounds more than this control allows. Set alignment volume allows user to set a volume offset, which can be a positive number. This number has to be passed as a parameter (in 0.01 dB). Get allows to read current alignment setting. Please, note that high level sounds may saturate with a positive alignment gain set.

`AUDSYS_COMMAND_SET_TRIM`

`AUDSYS_COMMAND_GET_TRIM` The trim controls are used to control the relative balance of the audio channels. The trim should be controlled during a speaker setup phase of system operation. When trim is configured, the parameter field of the config structure holds a pointer to an array of six 32-bit integers. The members of the array are ordered in the standard L, R, C, Sub, Lsur, Rsur order.

`AUDSYS_COMMAND_SET_CENTER_DELAY`

`AUDSYS_COMMAND_GET_CENTER_DELAY`

`AUDSYS_COMMAND_SET_SURROUND_DELAY`

`AUDSYS_COMMAND_GET_SURROUND_DELAY`

In the audio system, delays are specified in milliseconds. The maximum value is 30 ms. The minumum value is zero. Delays should be used to compensate for speaker positions in a room, and they should be adjusted in a system setup interface. The integer delay is cast to a Pointer and passed directly as the parameter field of the config structure.

`AUDSYS_COMMAND_SET_SAMPLERATE`

`AUDSYS_COMMAND_GET_SAMPLERATE`

The set and get sample rate commands pass the address of a floating point number.

`AUDSYS_COMMAND_MODIFY_SAMPLERATE`

The modify sample rate command passes the address of a floating point number that is multiplied by the currently set sample rate. This is designed to be used in systems where the clock is locked to an external source, such as the broadcast of MPEG audio and video. For example, the nominal sample rate is 48000. The MPEG clock recovery code has determined that the video clock should run at 27,000,010 Hz to regenerate the 90 kHz MPEG clock. The audio sample rate can be modified by 27,000,010/27,000,000 to compensate. In the end, this command will call **aoSetSrate** with the a value that is a floating point number times the nominal sample rate. It does not change the nominal sample rate, so subsequent modifications are still applied to the nominal sample rate. Since **aoSetSrate** uses the board support package, it is up to the board designer to ensure the function is efficient and callable from interrupts.

`AUDSYS_COMMAND_SET_BASS_REDIR_MODE`

The audio mixer supports bass redirection as described in the Dolby Licencee Information Manual. Version 1 of the mixer implements a set of second order crossover filters. The filter Q is 0.707, and the crossover frequency can be controlled with another command. This command allows the user to select one the values listed in under *Mixer Values* on page 278.

`AUDSYS_COMMAND_GET_BASS_REDIR_MODE`

The currently selected bass redirection mode (as listed above) is returned in the parameter field of the control structure.

`AUDSYS_COMMAND_SET_CROSSOVER`

`AUDSYS_COMMAND_GET_CROSSOVER`

Given in Hertz. Max value is 500. Min Value is 10. No bounds check is performed. The integer value is cast and passed directly in the parameter field of the config structure.

`AUDSYS_COMMAND_ENABLE_TONE_CONTROL`

`AUDSYS_COMMAND_DISABLE_TONE_CONTROL`

Enables or disables the tone control filter.

`AUDSYS_COMMAND_SET_BASS_GAIN`

`AUDSYS_COMMAND_GET_BASS_GAIN`

Sets/gets the tone control bass boost/cut filter. The parameter (float) values are given in dB in the range ±12 dB. Requires passing the pointer to the parameter.

`AUDSYS_COMMAND_SET_TREBLE_GAIN`

`AUDSYS_COMMAND_GET_TREBLE_GAIN`

Sets / gets the tone control treble boost/ cut filter. The parameter (Float) values are given in dB in the range ±12 dB. Requires passing the pointer to the parameter.

`AUDSYS_COMMAND_ENABLE_LOUDNESS_CONTROL`

`AUDSYS_COMMAND_DISABLE_LOUDNESS_CONTROL`

Enables or disables the loudness control filter. This filter has very little effect on loud sounds (close to 0 dBFS). For the soft sounds, the filter raises the low and high frequencies trying to compensate for the subjective feeling of week bass/treble of such soft sounds.

`AUDSYS_COMMAND_GET_LOUDNESS_MODE`

Passes the current setting of the loudness mode.

`AUDSYS_COMMAND_ENABLE_ECHO`

`AUDSYS_COMMAND_DISABLE_ECHO`

Enables/disables an echo effect to provide varying spaciousness. Stereo sounds only.

`AUDSYS_COMMAND_SET_ECHO_DELAY`

`AUDSYS_COMMAND_GET_ECHO_DELAY`
: Controls the delay of the rear speakers for the echo effect. Requires a pointer to the delay Float value in ms.

`AUDSYS_COMMAND_ENABLE_NOISE_SEQUENCER`

`AUDSYS_COMMAND_DISABLE_NOISE_SEQUENCER`
: During the room setup phase, the noise sequence can be used as a neutral source of sound for the adjustment of relative speaker levels. These commands take no parameters.

`AUDSYS_COMMAND_ENABLE_NOISE_SEQUENCER_ROTATE`
: Start the noise sequencer rotating its noise clockwise around the enabled channels. The noise sequencer must be enabled first. Rotation always starts in the center channel.

`AUDSYS_COMMAND_SET_NOISE_SEQUENCER_CHANNEL`
: If the noise sequencer is running, rotation is stopped and the output is forced to the channel specified. Specify the channel using the constants from tmalNoiseSeq.h, like NS_OUTCHAN_CENTER. Place the value in the parameter field of the control structure.

`AUDSYS_COMMAND_SET_TIMEDELAY`
: The parameter value contains the number of clock ticks to delay the audio against the video. The audio renderer ultimately treats this as a signed integer.

`AUDSYS_COMMAND_ENABLE_AV_SYNC_ADAPTATION`

`AUDSYS_COMMAND_DISABLE_AV_SYNC_ADAPTATION`
: Enable or disable the mechanism by which the audio renderer varies the AV Sync time constant to find an appropriate value. Useful as a debugging tool.

`AUDSYS_COMMAND_GET_AV_SYNC_TIMECONSTANT`

`AUDSYS_COMMAND_SET_AV_SYNC_TIMECONSTANT`
: Change the AV Sync time constant while the system is running. Pass the address of a floating point number.

`AUDSYS_COMMAND_GET_RANGE`

`AUDSYS_COMMAND_GET_STEP`
: These values allow a control program to use an integer scale for setting the current value of the master volume and the tone bass/ treble gains. Range returns a maximum and a minimum values on the integer scale. The controller program can operate on the integer values within this range. The real dB values to be passed to the audioSystem are n*step, where the step is a Float dB increment for the controls, and n is an integer between the minimum and the maximum values. The ‘get range’ command requires passing of a pointer to an Int minMax[2] array. First value in this array should be a control ID (**tsaAudSysControls_t**). In return the min/max values are placed in the table. The ‘get step’ command requires passing of a pointer to the control

| | |
|---|---|
| | ID (tsaAudSysControls_t). Float step value will be returned at this pointer. Currently only the master volume and bass/treble gains are supported. |
| `AUDSYS_COMMAND_SET_TM2_SPDIF_CHANN_PAIR` | |
| `AUDSYS_COMMAND_SET_TM2_SPDIF_CHANN_PAIR` | This command allows switching between the channel pairs during the SPDIF playback on the TM1300 or TM2. A value identifying the channel pair (0,1,2,or 3) has to be passed in the parameter. The 'Get' command returns the current setting. |

## Mixer Values

| | |
|---|---|
| `AMIX_bassRedirNone` | No bass redirection is applied. |
| `AMIX_bassRedirHPF_all_a` | As described on page 54 of the Dolby LIM, High pass filtering is applied to L, R, C, and surround outputs. All channels are summed to the LFE channel. It is assumed that 15 dB of gain will be added to the LFE channel in the analog output stage. This is Dolby mode 1. |
| `AMIX_bassRedirFullRangeLR_a` | The alternative form of Dolby mode 2, as described on page 55 of the Dolby LIM. Full range speakers are available for the front L and R channels. The other channels are high pass filtered at the crossover frequency. The low frequency energy (LFE) from the other channels is redirected into the LFE channel. The final summing of the LFE into the L and R channel (with associated gain) is assumed to be implemented in the analog output stage. |
| `AMIX_bassRedirFullRangeLRS_s` | Dolby mode 3, as described on page 56 of the Dolby LIM: Full range speakers are assumed for L, R, and surround channels. The LFE from the center channel is redirected. A subwoofer is assumed to be present. |
| `AMIX_bassRedirFullRangeLRS_ns` | Dolby mode 3, as described on page 56 of the Dolby LIM: Full range speakers are assumed for L, R, and surround channels. The LFE from the center channel is redirected. Gains are set as if no subwoofer is present. The subwoofer output is silent. |
| `AMIX_bassRedirExtern` | As described on page 71 of the Dolby LIM: All channels are given full range audio and the all channels are summed into the LFE channel. Bass redirection is assumed to be handled externally. |

## tsaAudSysStart

```
tmLibappErr_t tsaAudSysStart(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value as returned by **tsaAudSysOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

The audio system is started, using the setup specified in the setup function. All components are first stopped. Then they are connected as specified, and the necessary components are started. A task is created and used to monitor the aspects of the system that might change and necessitate a change to the setup.

## tsaAudSysStop

```
tmLibappErr_t tmalArendAORenderBuffer(
   Int   instance
);
```

### Parameters

| | |
|---|---|
| `instance` | Instance value as returned by **tsaAudSysOpen**. |

### Return Codes

| | |
|---|---|
| `TMLIBAPP_OK` | Success. |

### Description

Stop all components of the audio system.