Book 6—Audio Support Libraries

Part B:

Codecs

TriMedia

Version 2.0 beta

Table of Contents

Chapter 12 Dolby Digital AC-3 (AdecAc3) API

Chapter 12

Dolby Digital AC-3 (AdecAc3) 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. In addition, this algorithm is owned by Dolby Labs and an appropriate license must be obtained for its use. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

Dolby Digital AC-3 Standard Overview

Dolby Digital AC-3 is a digital compression standard for audio signals that was developed by Dolby Laboratories, Inc. It is a standard intended for use in high-quality, multichannel audio environments, but it also supports low bit-rate stereo or mono audio signal coding. Dolby Digital AC-3 application fields include digital television (DTV), sound on laser disk and digital versatile disk (DVD), as well as general multimedia PC or Internet applications. In addition to the pure compression feature required for efficient storage or transmission of audio data, AC-3 data streams contain information on the nature of the stream and conditions under which the data was recorded/sampled.

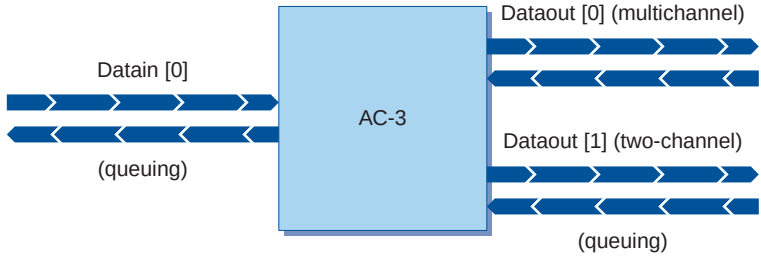


Figure 1 Structure of the Dolby Digital AC-3 Decoder

The standard supports a variety of different audio configurations. For example, an AC-3 decoder can map the incoming AC-3 bitstream onto the existing loudspeaker configuration by downmixing the input channels to the appropriate number of output channels.

Furthermore, AC-3 provides the means to adjust the dynamic range and apparent loudness of the decoded sound to compensate for various loudspeaker characteristics and listening environments. It is, for example, possible to maintain an equivalent loudness level when switching from one program to another in digital TV applications using AC-3 as audio standard.

A special mode allows for generating a Dolby Surround-compatible stereo signal. This is useful when someone wants to use the AC-3 decoder with an existing Dolby Surround or Dolby Pro Logic receiver/amplifier. In addition , AC-3 supports a special Karaoke mode.

The AC-3 standard supports data rates ranging from 32 kbps to 640 kbps and sampling frequencies of 32 kHz, 44.1 kHz and 48 kHz. The time resolution of the employed filter bank is 2.66 ms and the frequency resolution is 93.75 Hz at a sampling frequency of 48 kHz. According to Dolby, audio transparency is achieved at 320 kbps and upward in 5.1 channel mode. Dolby distinguishes between three different quality levels of AC-3 decoder implementations ranging from 16-bit through 20-bit resolution of the generated PCM samples.

AC-3 data streams contain up to five independently coded full-bandwidth channels and one low-frequency effects channel. shows an AC-3 audio environment that is set up with five conventional speakers and one subwoofer for bass effects. For more information,

refer to http://www.dolby.com or http://www.atsc.org. The AC-3 standard is document A/52 in the terminology of the Advanced Television System Committee.

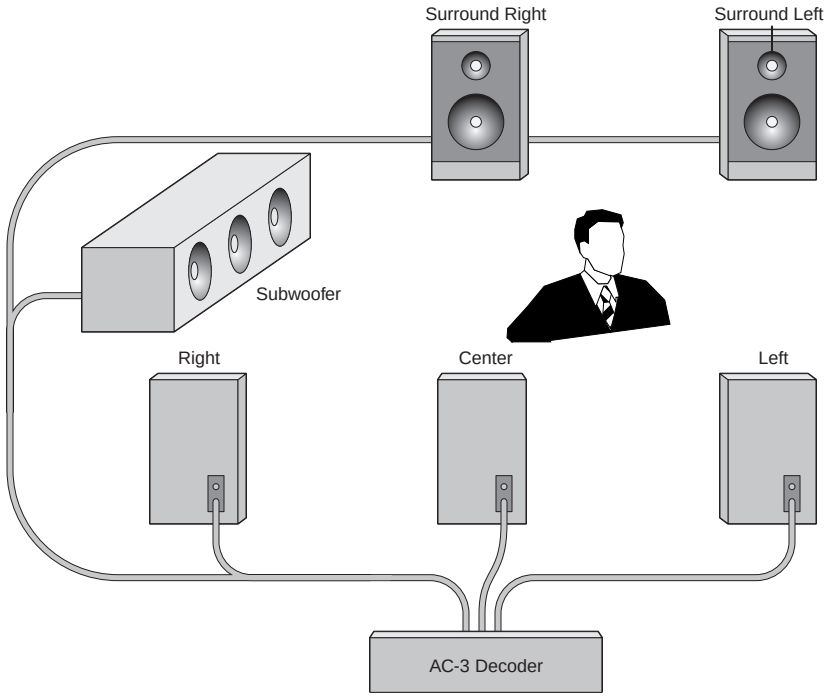


Figure 2 AC-3 audio environment

Composition of AC-3 Streams

The AC-3 bitstreams consist of frames that are coded independently of one another (see). Each frame represents 1,536 PCM samples across all coded channels. This means that if a sample rate of 48 kHz is applied, each frame represents 32 ms audio. All frames are of equal size for 32 kHz and 48 kHz modes. In 44.1 kHz mode, a difference in length of 2 or 4 bytes between successive frames is possible. This is necessary because the data rate divided by the number of frames per second cannot be represented by an integer number for some supported data rate values.

Sync	CRC #1	SI	BSI	Audio Block 0	Audio Block 1	Audio Block 2	Audio Block 3	Audio Block 4	Audio Block 5	CRC #2

Figure 3 Frame structure

Each frame begins with a bitstream information field that contains general information (such as sample rate and channel configuration), and special information regarding the coding strategies. The actual audio samples are located in 6 audio blocks, each of which represents 256 samples across all coded channels. The structure of the audio blocks is shown in . In addition to the audio blocks and bitstream information, two cyclic redundancy checks (CRC) are placed in each frame in order to detect transmission errors.

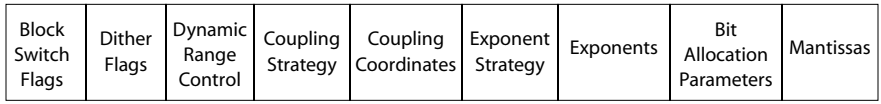


Figure 4 Audio block structure

The audio blocks contain 256 coded frequency coefficients for each coded audio channel. These are used as input for an inverse frequency transform. The encoder splits the frequency samples into exponents and mantissas, as is common in floating-point representation. The exponents are coded with fixed data lengths. Bit-allocation for the mantissas is calculated by the decoder by using the exponents and additional bit allocation parameters. This adaptive coding approach enables higher compression ratios than fixed bit-allocation coding. Even higher compression can be achieved by sharing high-frequency carrier components across channels, a technique called channel coupling.

AC-3 Decoding Scheme

illustrates the AC-3 decoding process. The bitstream information decoding block extracts information from the bitstream. This is used within the actual decoding process by the computational blocks, and also by the application to correctly set up the audio environment. Frame information (such as sample rate) and audio block information (such as coding strategies) is obtained at this stage.

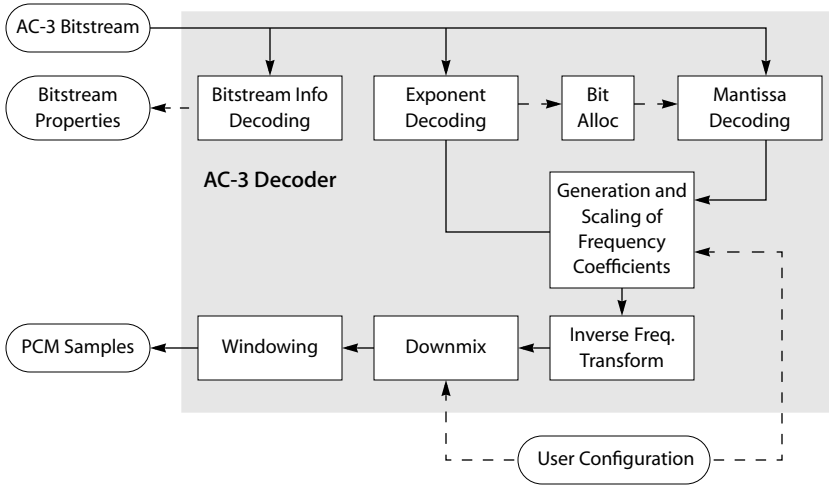


Figure 5 AC-3 Decoder data flow block diagram showing the functions blocks of the decoder core. Dotted arrows represent control data and solid arrows represent the AC-3 bitstream itself, as well as decoded samples in the time and frequency domain.

The next step in the AC-3 decoding process is unpacking the exponents. Exponent coding is fixed, which means that they can be extracted by using the exponent strategy field and fixed tables known to both encoder and decoder. The exponents are coded in groups by 7-bit data words. After the exponents are decoded, the bit allocation for the mantissas is computed, using bit-allocation information and the exponents. Calculation of the bit allocation is done, in principle, by using a psycho-acoustic model, and is performed in seven successive steps. Within these steps, the masking curve used by the encoder is derived from an estimate of the log-spectral envelope of the audio block. It is obtained from the values of the exponents. The resulting mantissas are then dequantized and denormalized. The resulting frequency coefficients are in fixed-point format, which can then be scaled with respect to the user's requirements and compression parameters supplied in the bitstream.

The actual inverse transform is a so-called Inverse Modified Discrete Cosine Transform (IMDCT), which implements a time domain alias cancellation (TDAC) synthesis filter bank. Refer to the literature for more information on this filter bank.

The next step in the decoding process is downmixing to an appropriate number of output channels. This is necessary if the number of encoded channels differs from the number of loudspeakers connected to the decoder. Karaoke processing is also performed within the downmixing if it is enabled.

Once the downmixing is complete, windowing is applied for anti-aliasing the audio signal. The last step is to overlap the first half of the windowed block with the second half of the previous block. The overlapped MDCT of the encoder necessitates this step.

AC-3 Synchronization Scheme

As previously stated, an AC-3 decoder operates on a frame by frame basis. The length of an encoded audio frame is between 128 bytes and 3,840 bytes, depending on the sampling frequency and the data rate. It always starts with a 16-bit sync word. An AC-3 decoder must find a sync word prior to doing the actual decoding described in the previous section. A description of the structure of an AC-3 frame can be found in the earlier section *Composition of AC-3 Streams*.

Together with the sync word, two CRC words are transmitted in every AC-3 frame. The first one checks the first 5/8 of the frame's data and the second one checks the rest. If one CRC fails, the respective part of the frame will not be decoded and a special error concealment algorithm will be applied instead. The worst-case situation from the decoder perspective is if a corrupted frame conveys data that passes both CRC tests and has a valid sync word. In this case, the decoding algorithm can suffer from misinterpretation problems. By design, the probability of the occurrence of such a situation is quite low, (for example, if the data rate is 384 kbps, the probability of false detected sync word accompanied by successful CRC calculations is 0.000035%, which is once in 26 hours of decoding).

In cases where synchronization or transmission problems occur during processing, an AC-3 decoder conceals the error either by muting or by repeating the last correctly decoded audio block. Dolby recommends repeating rather than muting for a certain number of consecutively damaged blocks. If further frames are corrupted, muting may be applied. The nature of the overlap add window in the last processing stage of the decoder supports this type of error concealment; the TriMedia AC-3 decoder library supports this model. An application programmer integrating the AC-3 decoder library into an application can choose the maximum number of repeated blocks in a row before muting is applied.

TriMedia AC-3 API Overview

The TriMedia AC-3 decoding library is fully hardware-independent, thereby providing the highest flexibility. Applications using the API must implement any hardware-specific processing, particularly the input of AC-3 data and output of PCM data. Philips provides a set of interface libraries for this purpose. The AC-3 data source could be, for example, the PCI bus, the audio input interface, or, in special environments, the video input interface. It could also be, of course, the output of another data-processing component, such as an MPEG-2 demultiplexer.

illustrates the AC-3 decoding library input and output streams. The TriMedia AC-3 decoder library implements a data-processing filter that has one input and two outputs. It expects compressed data at its input and produces up to 6 channels of PCM samples at the Main Multichannel Audio output. A Secondary Stereo output is optional, and can be used in special consumer electronics application environments. It conveys either

decoded stereo PCM samples or the uncompressed AC-3 data. It is possible to apply a Dolby Surround Pro Logic compatible downmixing scheme. This data can be used, for instance, to drive a headphone output. In the second case, the output provides the AC-3 data itself in a special format (IEC61937, also referred to as 1937 format in this document), which supplies the AC-3 data at audio data rates and can be used to interface with external decoders.

Aside from the pure data processing, the AC-3 library has functions that enable the configuration of the decoder, (for example, number of output channels), and functions that return properties of the compressed audio, (for example, the sampling rate). These properties are required to properly set up the audio-reproduction environment.

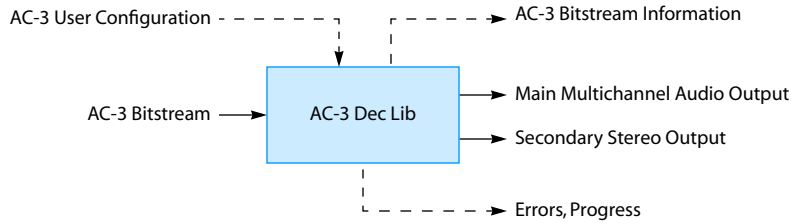


Figure 6 AC-3 library as data processing block. Solid arrows represent the input and output data streams. Dashed arrows represent control data used to set up the decoder itself and the environment of the decoder.

The TriMedia AC-3 decoding library implements the standard TSA API, which enables application programmers to easily implement frame-oriented, as well as streaming-data, processors. This API consists of two layers: the operating system-dependent OL layer and the operating system-independent AL layer. In the following discussions, the OL layer is also referred to as the OS library. The AL layer is also known as the AL library.

The reason for having two different library front ends (which actually are layered on each other) is to provide the applications programmer with flexible interfaces on different abstraction levels suitable for different application requirements. Philips recommends that applications use the OL layer, rather than the AL layer, since the OL layer supplies automatic data transmission and reception using a message-passing scheme. The advantage is that applications connecting multiple OL libraries can be programmed easily and quickly. The disadvantage is that a certain amount of overhead is introduced, which consumes additional CPU time and memory.

Both forms of the AC-3 decoder library have a number of similarities. Both offer the seven standard API functions (GetCapabilities, Open, Close, GetInstanceSetup, InstanceSetup, Start, and Stop) with the name prefix tmolAdecAc3 or tmalAdecAc3, respectively. In addition, the AC-3 library provides functions for operation in the so-called push or non-streaming mode.

The AC-3 decoder works with instances. Multiple AC-3 decoders can be run at the same time being identified by their instance. The current TriMedia AC-3 decoder library can be instantiated up to ten times.

Both library forms are implemented following the principle of separating the pure data-stream processing from the buffer handling, from control and command processing, and also from error and progress reporting. Also, OS-dependent operations, (such as dynamic memory allocs) are completely separated from the data-processing functions implemented in the AL library.

An application programmer must decide whether to use the AL layer or the OL layer. Implementing an application at the OL layer is easier and requires less development time; however, there is an overhead cost (CPU load and memory requirements). On the other hand, applications implemented using the AL library in non-streaming mode might be faster. The disadvantage is that the programmer must now implement synchronization and data packet processing which is otherwise masked by the streaming mode APIs.

The furnished API does not match the interface format recommended by Dolby in their DSP Software Interface Protocol document. This is because the Dolby interface does not optimally match a media processor such as the TriMedia. It is intended for pure DSP development. However, the same functionality is exposed by the TriMedia API. The names of bitstream information fields and decoder configuration parameters are identical to those specified in Dolby's interface. Therefore, no problems are anticipated for programmers, (who are used to working with the DSP Software Interface Protocol) to develop applications using the TriMedia AC-3 decoder library.

The AL Layer

The operating system-independent library can be used in two different modes: *push mode* or *pull mode*. While the pull mode supports the streaming data model, the push mode operates at the granularity of a single data frame.

In the push model, the AC-3 decoder library acts passively. This means it does not actively request data or return data. All decoding actions are controlled by the application built above the AC-3 decoder library. All buffer management and synchronization issues must be managed by the application. shows the order in which the functions provided by the AL library are typically called when the decoder application is working in

push mode (non-streaming). The operation mode of the AL library is selected within the instance setup function. It is determined by a field of the instance setup struct.

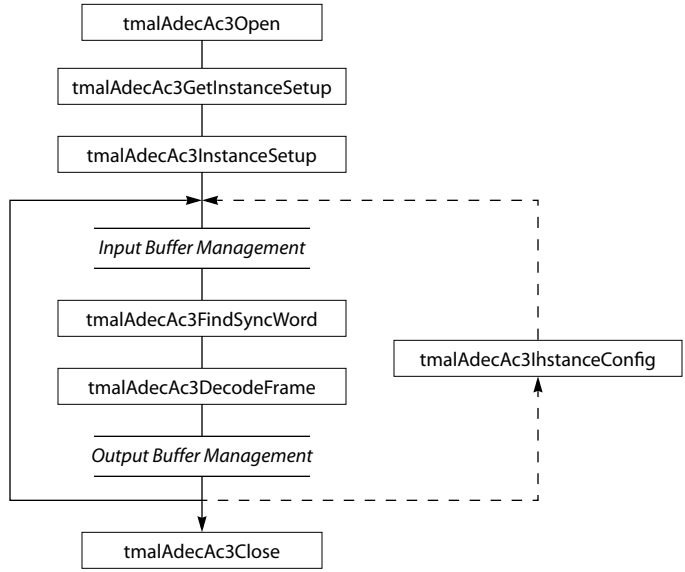


Figure 7 Operation of an AL decoder in push mode. Shaded function blocks are to be implemented by the application programmer. White blocks represent API functions of the TriMedia AC-3 decoder library. Note that this is a simplified block diagram.

First an instance of the decoder is obtained by **tmalAdecAc3Open**. The next step is the configuration of the decoder instance. With **tmalAdecAc3GetInstanceSetup** the application obtains a pre-configured setup structure. This structure is then modified by the application and returned to the decoder by calling **tmalAdecAc3InstanceSetup**. This configures the decoder. After that, a loop **tmalAdecAc3FindSyncword** and **tmalAdecAc3-DecodeFrame** is executed. If changes to the configuration of the decoder are required, the function **tmalAdecAC3InstanceConfig** can optionally be called. After finishing, the application must finalize the decoder instance with **tmalAdecAc3Close**.

Using the AL library in pull mode results in the following order of library function calls shown in . The path of the arrows shows the handling of the input and output data streams and the control processing as error concealment which must be implemented by

the application programmer. Note that **tmalAdecAc3DecodeFrame** is referred to as a ProcessData function in the TriMedia Software Architecture.

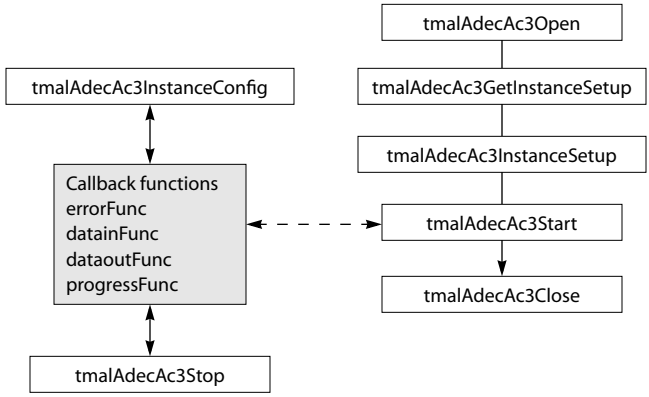


Figure 8 Operation of an AL decoder in pull mode. Shaded function blocks are the callback functions to be implemented by the application programmer. White blocks represent API functions of the TriMedia AC-3 decoder library. Note that this is a simplified block diagram.

The first steps in using the AL AC-3 decoder API in pull mode are the same as in push mode: the instance of the decoder is obtained and then initialized. Once the decoder is started by calling **tmalAdecAc3Start**, it runs in streaming mode until the function **tmalAdecAc3Stop** is called either from one of the callback functions, an interrupt routine, or a task running concurrently. The call of **tmalAdecAc3Stop** sets a stop flag, which is checked by **tmalAdecAc3Start**. After leaving **tmalAdecAc3Start**, the current instance of the AC-3 decoder must be finalized by calling **tmalAdecAc3Close**. While running the decoder can be reconfigured by calling **tmalAdecAc3InstanceConfig** from either a callback function, an interrupt service routine, or a different task if an operating system is used.

The function **tmalAdecAc3Start** does all input and output data processing by calling callback functions. The application programmer must supply callback functions for the AC-3 decoder running in the AL pull mode. A more detailed description of the behavior of **tmalAdecAc3Start** is given in a later section.

The OL Layer

The OL layer works in a similar fashion as the AL layer in pull mode. Unlike the AL layer, default data-processing is based on message queues implemented using ISI's pSOS+™ operating system. As a result, an application programmer using the OL layer must simply set up the appropriate queues, and then connect inputs and outputs of each of the com-

ponents along a processing chain. shows a block diagram describing the order of function calls in an application using the OL layer.

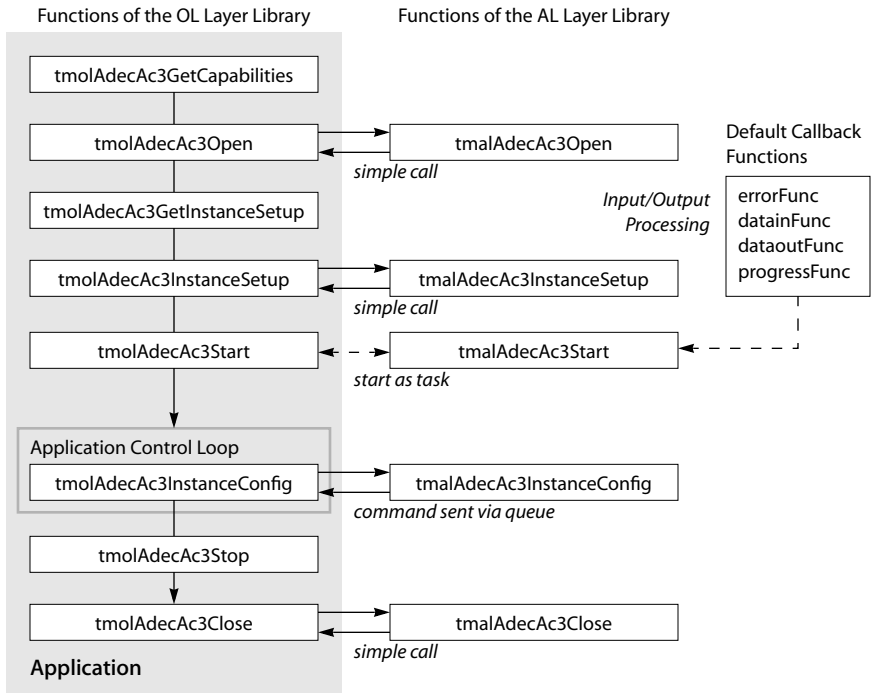


Figure 9 Operation of an OS decoder. White blocks represent functions provided by the library. Functions on the left side are provided by the OS library; functions on the right side belong to the AL library.

The creation of queues and allocation of packet memory can be done using the function **tsaDefaultInOutDescriptorCreate** which is part of the Default library. Similarly, the resources required for the control queue processing can be acquired by the application using **tsaDefaultControlDescriptorCreate**.

As shown in , some of the OS layer library functions implement their functionality by delegating to their respective AL layer library counterparts. The open and instance setup functions do this directly by the use of a function call. A difference between AL layer and OL layer programming is that an OL application must query for the library's capabilities in order to negotiate with connected components about supported TSA packet formats. After setup and configuration of the decoder, **tmalAdecAc3Start** is started as a pSOS task created by **tmolAdecAc3Start**. The decoder core function **tmalAdecAc3Start** then runs parallel to the user application program. A decoder configuration change is possible after the start of the decoder by calling **tmolAdecAc3InstanceConfig**. This function sends a command via the command queue to its AL layer counterpart which then performs the requested configuration change. Since the command queue is only read when data input

or output is performed, command execution is asynchronous. Command execution occurs in the context of the decoder, while **tmolAdecAc3InstanceConfig** executes in the context of the application. The user application can stop the AC-3 decoder by calling **tmolAdecAc3Stop** which causes the decoder core function **tmalAdecAc3Start** to flush all data packets that are kept and return. After that the associated task is destroyed. Note that data still remaining in the input queue is returned unprocessed in the empty queue.

Configuring the Decoder

Before the decoding of an AC-3 bitstream can be started the decoder library needs to be configured. This holds for both, streaming and non-streaming modes. Both modes supply a GetInstanceSetup and an InstanceSetup function. The purpose of the InstanceSetup functions is to provide the decoder with all required information of the data it will receive at its input and what sort of data it is expected to send at its outputs. In addition to this, the decoder receives all necessary configuration properties such as downmix settings and dynamic range parameters.

Setup of an OL Layer Decoder Application

This section describes the setup of the AC-3 library with a sample application. It connects the AC-3 decoder at its input side to the TriMedia File Reader component, and on its output side to the TriMedia Audio Renderer component. The code is slightly modified from the example program *exolAdecAc3.c*.

```
void tmosMain(){
   tmLibappErr_t                rval;
   Int                          readerInstance, decoderInstance,
                                arendInstance;
   ptmolFreadInstanceSetup_t    readerSetup;
   ptmolAdecAc3InstanceSetup_t  decoderSetup;
   ptmolArendAOInstanceSetup_t  arendSetup;
   tmAudioFormat_t              audioFormat;
   tsaInOutDescriptorSetup_t    iodSetup;
   ptsaInOutDescriptor_t        iodReaderDecoder;
   ptsaInOutDescriptor_t        iodDecoderRenderer;
   ptmolFreadCapabilities_t     frCaps;
   ptmolArendAOCapabilities_t   arCaps;
   ptmAdecAc3Capabilities_t     ac3Caps;
   ptsaControlDescriptor_t      decoderCommand;
   tsaControlDescriptorSetup_t  csetup;
   char                         FileName[80];
   tmosInit();
```

The above code segment shows the declarations of the variables used by the following sample code. The operating system is initialized by **tmosInit**.

At first instances of the three involved components are acquired with the respective Open functions. In addition, the components' capabilities are obtained. They will be used for the initialization of the queue packet formats.

```
rval = tmolFreadOpen(&readerInstance);     /* open file reader    */
rval = tmolAdecAc3Open(&decoderInstance);  /* open decoder        */
rval = tmolArendAOOpen(&arendInstance);    /* open audio renderer */
tmolFreadGetCapabilities(&frCaps);
tmolArendAOGetCapabilities(&arCaps);
tmolAdecAc3GetCapabilities(&ac3Caps);
```

The next step is the configuration of the format of the packets circulating between the File Reader and the AC-3 Decoder. It is AC-3 data in raw format. Note that a **dataSubtype** must be specified. The format manager requires this.

```
audioFormat.size           = sizeof(tmAudioFormat_t);
audioFormat.hash           = 0;
audioFormat.referenceCount = 0;
audioFormat.dataClass      = avdcAudio;
audioFormat.dataType       = atfAC3;
audioFormat.dataSubtype    = apfGeneric;
audioFormat.description    = 0;
audioFormat.sampleRate     = 0.0;
```

This format is used for the creation of an input/output descriptor which is shared between the File Reader and the AC-3 Decoder. In this configuration, the AC-3 decoder is the receiver of packets. The **receiverIndex** field contains the number of the input pin which receives data packets after startup. Since the AC-3 decoder has only one input pin, the value is always **ADECAC3_MAIN_INPUT**. The input/output descriptor pointer **iodReaderDecoder** obtained from the descriptor creation function is used later on for the setup of the File Reader and AC-3 Decoder. The memory for the descriptor is allocated within **tsaDefaultInOutDescriptorCreate**.

```
iodSetup.format          = (ptmAvFormat_t)&audioFormat;
iodSetup.flags           = tsaIODescSetupFlagCacheMalloc;
iodSetup.fullQName       = "ac3f";
iodSetup.emptyQName      = "ac3e";
iodSetup.queueFlags      = tmosQueueFlagsStandard;
iodSetup.senderCap       = frCaps->defaultCapabilities;
iodSetup.receiverCap     = ac3Caps->defaultCapabilities;
iodSetup.senderIndex     = FREAD_MAIN_OUTPUT;
iodSetup.receiverIndex   = ADECAC3_MAIN_INPUT;
iodSetup.packetBase      = 0;
iodSetup.numberOfPackets = MAX_PACKETS;
iodSetup.numberOfBuffers = 1;
iodSetup.bufSize[0]      = CBUFSIZE;
rval = tsaDefaultInOutDescriptorCreate(&iodReaderDecoder, &iodSetup);
```

On its output side, the AC-3 decoder is connected to the Audio Renderer. The **senderIndex** field is set to **ADECAC3_MULTICHANNEL_OUTPUT**. The second optional output pin of the AC-3 decoder is not used in this sample application. It is important to set the **tsaIODescSetupFlagCacheMalloc** flag, because otherwise cache coherency problems might occur with the audio output hardware. The input/output descriptor pointer **iodDecoderRenderer** is used later on for the setup of the AC-3 decoder and the Audio Renderer. Note that no format is specified by this input/output descriptor setup struct. However, the

queues between the decoder and the renderer supply PCM audio data at a particular sample rate. This sample rate value, however, is not available before the decoder starts. It is encoded within the incoming AC-3 bitstream. Hence, an appropriate format for this descriptor is installed by the decoder when it starts.

```
iodSetup.format          = Null;
iodSetup.flags           = tsaIODescSetupFlagCacheMalloc;
iodSetup.fullQName       = "pcmf";
iodSetup.emptyQName      = "pcme";
iodSetup.queueFlags      = tmosQueueFlagsStandard;
iodSetup.senderCap       = ac3Caps->defaultCapabilities;
iodSetup.receiverCap     = arCaps->defaultCapabilities;
iodSetup.senderIndex     = ADECAC3_MULTICHANNEL_OUTPUT;
iodSetup.receiverIndex   = ARENDAO_MAIN_INPUT;
iodSetup.packetBase      = 100;
iodSetup.numberOfPackets = MAX_PACKETS;
iodSetup.numberOfBuffers = 1;
iodSetup.bufSize[0]      = DBUFSIZE;
rval = tsaDefaultInOutDescriptorCreate(&iodDecoderRenderer,&iodSetup);
```

Once the input/output descriptors for the system are created, the setup of the individual components can take place. The File Reader is now set up as the first component in the processing chain.

```
sprintf(FileName, "cave.ac3");
rval = tmolFreadGetInstanceSetup(readerInstance,&readerSetup);
readerSetup->defaultSetup->outputDescriptors[FREAD_MAIN_OUTPUT]
                                          = iodReaderDecoder;
readerSetup->fileName                     = FileName;
readerSetup->defaultSetup->priority       = READER_PRIORITY;
rval = tmolFreadInstanceSetup(readerInstance,readerSetup);
```

For the initialization of the AC-3 decoder, a control descriptor is required. After the decoder is started, the associated control queue can be used by the application to either change internal settings of the decoder or obtain settings from the decoder. The function **tsaDefaultControlDescriptorCreate** allocates this control descriptor and returns a pointer to it as its first argument.

```
csetup.commandQName  = "ac3C";
csetup.responseQName = "ac3R";
csetup.queueFlags    = tmosQueueFlagsStandard;
csetup.flags         = 0;
rval = tsaDefaultControlDescriptorCreate(&decoderCommand,&csetup);
```

The AC-3 decoder is the next component to be configured. First a pre-configured instance setup struct is obtained from the decoder component via **tmolAdecAc3GetInstanceSetup**.

```
rval = tmolAdecAc3GetInstanceSetup(decoderInstance,&decoderSetup);
```

Certain fields of the returned instance setup struct must be updated. Pointers to the two input/output descriptors and to the control descriptor are stored in the instance setup

struct. The secondary stereo output pin is then disabled by assigning a Null pointer to its output descriptor.

```
decoderSetup->defaultSetup->inputDescriptors[ADECAC3_MAIN_INPUT]
                                                    = iodReaderDecoder;
decoderSetup->defaultSetup->outputDescriptors[ADECAC3_MULTICHANNEL_OUTPUT]
                                                    = iodDecoderRenderer;
decoderSetup->defaultSetup->outputDescriptors[ADECAC3_TWO_CHANNEL_OUTPUT]
                                                    = Null;
decoderSetup->defaultSetup->controlDescriptor = decoderCommand;
```

After the installation of the control and I/O descriptors, the callback function which are implemented in the application must be installed. In this application a progress and an error function are implemented as callback functions. The progress function reports when a major change in the parameters of the incoming bitstream occurs. Refer to page 45 for more information on the progress function and its flags.

Another important field contains the priority value of the component. The priority should be chosen with care because it affects the stability and performance of the application.

```
decoderSetup->defaultSetup->progressReportFlags = A3_PROG_REPORT_CHANGES;
decoderSetup->defaultSetup->progressFunc        = decoderProgressFunc;
decoderSetup->defaultSetup->errorFunc           = ac3_error_func;
decoderSetup->defaultSetup->priority            = DECODER_PRIORITY;
```

The decoder requires information about the output format to be exposed by its output pins. It gets this information via the **tmolAdecAc3InstanceSetup_t** fields **pcmFormatOut0**, **precisionOut0**, **formatOut1**, and **precisionOut1**. See section *AdecAc3 Inputs and Outputs* for detailed information on the supported configurations. In this particular application only the **pcmFormatOut0** field is used because the **precisionOut0** field is redundant when the packet format is a 16-bit PCM format; **formatOut1** and **precisionOut1** do not matter because the second output is not used.

```
decoderSetup->pcmFormatOut0  = apfFiveDotOne16;
```

The configuration of the decoder's data and control interfaces is finished with this step in the example program. No modifications are made to the internal decoder configuration which controls the actual signal processing. It can be altered by changing the settings of the config structure. A pointer to this config structure is an element of the instance setup struct. A downmix to stereo could for instance be switched on by:

```
decoderSetup->config->outputMode  = A3_OUTCHANCONFIG_2_0;
```

Now that all necessary fields of the instance setup struct are updated the decoder setup function can be called.

```
rval = tmolAdecAc3InstanceSetup(decoderInstance, decoderSetup);
```

The last component to be configured is the Audio Renderer which is connected to the main multichannel output of the AC-3 decoder. Its input descriptor is shared with the AC-3 decoder's main multichannel output. Note that the packet size **DBUFSIZE** of the

PCM packets must be a multiple of 64 bytes. Otherwise the audio output device library would assert.

```
arendSetup->defaultSetup->inputDescriptors[ARENDAO_MAIN_INPUT]
                                                    = iodDecoderRenderer;
arendSetup->defaultSetup->errorFunc                 = arend_error_func;
arendSetup->defaultSetup->priority                  = AREND_PRIORITY;
arendSetup->operationalMode                         = AR_MODE_CONSERVATIVE;
arendSetup->maxBufferSize                           = DBUFSIZE;
rval = tmolArendAOInstanceSetup(arendInstance,arendSetup);
```

Finally, all components are configured and ready to start. In principle, the order of starting is not important. However, it is recommended to start the component first that receives data last.

```
rval = tmolArendAOStart( arendInstance   );
rval = tmolAdecAc3Start( decoderInstance );
rval = tmolFreadStart  ( readerInstance  );
```

After stopping and closing the components a clean up of the two I/O descriptors and the control descriptor is necessary.

Setup of an AL Decoder Application

This section describes how the AC-3 decoder must be configured to be used in non-streaming mode at the AL layer. The source code fragments used below are taken from the example program *exalAdecAc3ns.c*, which implements a file based AC-3 decoder. It reads compressed AC-3 data from an input file, performs input data buffer management, calls the core decoder on a frame by frame basis, and stores the decoded six channel PCM samples in the output file *output.pcm*. The application configures the input pin and the main multichannel output pin for this purpose.

The following structure variables are used during setup:

```
static tsaInOutDescriptor_t    idesc, odesc0, odesc1;
static ptsaInOutDescriptor_t   inputDescriptors[1], outputDescriptors[2];
static tmAudioFormat_t         ac3Format, pcmFormat;
ptmalAdecAc3InstanceSetup_t    ac3Setup;
Int                            status;
```

Note that static memory is allocated for the I/O descriptors. At the OL layer a default helper function is used for creating I/O descriptors. This function is not available to AL layer applications. It is the responsibility of the application to provide a set of properly allocated I/O descriptors for the setup of the AC-3 decoder library.

The application needs to supply one format struct for the decoder input and one for the decoder output. The format for the input side is raw data mode AC-3. In contrast to the OL layer example, the value of the **dataSubtype** field does not matter in the AL example, when the **dataType** is **atfAC3**.

```
ac3Format.size              = sizeof( tmAudioFormat_t);
ac3Format.dataClass         = avdcAudio;
ac3Format.dataType          = atfAC3;
```



```
ac3Format.dataSubtype       = 0;
ac3Format.description       = 0;
ac3Format.sampleRate        = 0.0;
```

For the decoder output a six channel 16-bit precision PCM format is prepared.

```
pcmFormat.size              = sizeof( tmAudioFormat_t);
pcmFormat.dataClass         = avdcAudio;
pcmFormat.dataType          = atfLinearPCM;
pcmFormat.dataSubtype       = apfFiveDotOne16;
pcmFormat.description       = 0;
pcmFormat.sampleRate        = 0.0;
```

Next, a pre-configured instance setup structure must be obtained with the AL layer **Open** and **GetInstanceSetup** functions.

```
status = tmalAdecAc3Open(&instance);
status = tmalAdecAc3GetInstanceSetup( instance, &ac3Setup);
```

The setup of the AC-3 decoder in non-streaming mode is a little bit more complex than the setup of an OL layer decoder because the application has to create manually input/output descriptors similar to those created by **tsaDefaultInOutDescriptorCreate**. This is achieved by following code fragment. Pointers to the statically allocated input/output descriptor pointer arrays are assigned to the respective fields of the instance setup struct (first two lines of the following code fragment). This is necessary, because the current implementation of **tmalAdecAc3GetInstanceSetup** does not allocate the memory for these pointer arrays. However, this will be changed in future releases.

In non-streaming mode no strict format handling is required. The only fields that matter for the input format are **dataClass** (always **avdcAudio**) and **dataType** (**atfAC3** or **atf1937**). In the main multichannel output format, the fields that matter are **dataClass** (always **avdcAudio**), **dataType** (always **atfLinearPCM**), **dataSubtype**, and description (if 32-bit **dataSubtype** is chosen). The second output is never used in non-streaming mode.

At the OL layer, a component connected to an output of the AC-3 decoder determines the sampling rate by the format of the outgoing PCM packets. In non-streaming mode, the decoder returns this information to the caller in its argument struct, refer to 1 and 1.

```
ac3Setup->defaultSetup->inputDescriptors         = inputDescriptors;
ac3Setup->defaultSetup->outputDescriptors        = outputDescriptors;
ac3Setup->defaultSetup->inputDescriptors[ADECAC3_MAIN_INPUT]
                                                 = &idesc;
ac3Setup->defaultSetup->outputDescriptors[ADECAC3_MULTICHANNEL_OUTPUT]
                                                 = &odesc0;
ac3Setup->defaultSetup->outputDescriptors[ADECAC3_TWO_CHANNEL_OUTPUT]
                                                 = NULL;
idesc.format                                     = (ptmAvFormat_t)&ac3Format;
odesc0.format                                    = (ptmAvFormat_t)&pcmFormat;
ac3Setup->libraryMode                            = A3_LIB_MODE_PUSH;
```

The instance setup struct is now configured and can be used to setup the decoder.

```
status = tmalAdecAc3InstanceSetup(instance, ac3Setup);
```

After the configuration is finished the actual decoding can be performed. Refer to Figure 11 on page 28 and to *The AL Layer* on page 16 for further information.

Implementation Aspects

This section describes some implementation aspects of the AC-3 decoder library, such as granularity of processed data, decoder delays, formats of input and output data, along with general hints for application programmers and examples of configurations.

Frame versus Block-Oriented Decoding

You must not be concerned with inner-frame-processing, such as CRC calculations and handling of the audio blocks. This section describes the granularity of the data processed by the AC-3 decoder library and provides an overview of some implementation details. Another important issue discussed in this section is decoding delay.

The decoder delay of a complete AC-3 system is defined as the time between arrival of the first byte of the compressed data and the display of the first decoded PCM sample (see Figure 10).

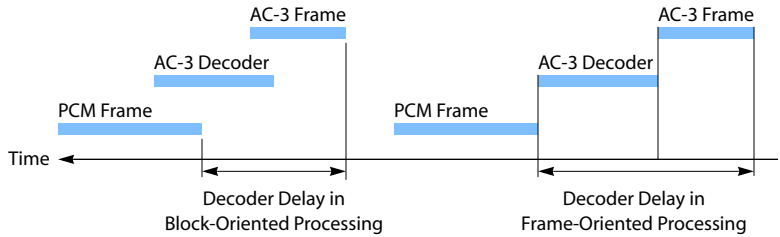


Figure 10 Decoding delay of frame-oriented and block-oriented decoder applications.

Different application may have completely different delay requirements. These requirements are dominated primarily by synchronization issues. AC-3 will mostly be used where video and lip synchronization must be maintained.

The difference between a frame and a block-oriented decoder is the moment in time when the decoder starts decoding of the incoming AC-3 data. The frame-oriented decoder starts decoding after a complete frame of AC-3 data has been received. Normally, a frame-oriented decoder would be able to deliver first PCM samples after decoding of the first audio block. In the TriMedia implementation, the AC-3 decoder returns the 1,536 PCM samples of a decoded frame as a single entity instead of as a series of 256 PCM samples blocks. The frame-oriented decoding is only supported by the AL library in push mode.

A block-oriented decoder can start to decode after 5/8 of an AC-3 frame is received. This is possible because 5/8 of a frame is exactly the range covered by the first CRC word. It is guaranteed by the standard that this first part of the frame contains at least two complete audio blocks. Thus, the first two blocks can be decoded while still receiving the remainder of the compressed data. In data-streaming mode (pull mode), the TriMedia AC-3 decoder library implements block-oriented decoding.

If the AC-3 library is intended to be used in an AC-3 decoder-only application, it must be ensured that Dolby's latency requirements are exactly met. Dolby defines the decoder latency as follows:

```
Latency = (time to read 2/3 of a frame) + (Block Period)
```

The block period is the time period of one audio block and, therefore, dependent on the sampling frequency:

Sampling Frequency	48.0 KHz	44.1 KHz	32.0 KHz
Block Period	5.33 ms	5.80 ms	8.00 ms

Dolby chose 2/3 instead of 5/8 of a frame as the reference time until the decoding of the frame can be started because this ratio is easier to handle. The time required to read the data depends on the transmission mode. In continuous transmission mode, the time derives from the block period as follows:

```
(time to read 2/3 of a frame) = 4 * (Block Period)
                    => Latency = 5 * (Block Period)
```

In contrast to continuous transmission mode, the time to read 2/3 of a frame in burst transmission mode depends on the data rate of the transmission channel and on the AC-3 bitstream data rate. Dolby considers only S/P DIF digital audio connections for such transfers. S/P DIF is a digital audio interface standard normally used to convey stereo PCM samples. The standard has been extended to convey compressed audio as well, but still operating at normal audio data rates. Since the required bandwidth for compressed audio is less than for uncompressed audio, the S/P DIF bitstream is split into blocks containing relevant data and blocks containing stuffing bits. The blocks containing relevant data are called *bursts*. A burst contains exactly one single frame of AC-3 data. The temporal distance between the starting points of two successive bursts corresponds to the time period a burst represents (32.00, 34.83, or 48.00 ms). This time is the derived from the number of samples per AC-3 frame divided by the sampling frequency. The time required to read 2/3 of the frame is computed from the S/P data rate and from the size of the AC-3 frame in 16-bit words, which is a function of sampling frequency and data rate. The latency range is from 5.78 ms (32 kbps at 48 kHz) through 28.00 ms (640 kbps at 32 kHz). Fore more information on latency issues refer to the document *Dolby AC-3 Multichannel Digital Audio Decoding System For Consumer Products, Version 1* as of October 10, 1995.

The data processing implemented by the push model or non-streaming TriMedia AC-3 decoder API is frame-oriented. That means the application built upon the library must ensure that **tmalAdecAc3DecodeFrame** always gets a complete AC-3 frame within its parameters struct. The application is responsible for the acquisition AC-3 data, setting up the struct, and checking with help of **tmalAdecAc3FindSyncword** if a complete frame is in the data referred to by the struct (**ac3Packet->buffers[0].data field**). Before calling **tmalAdecAc3DecodeFrame**, you must ensure that **ac3Packet–>buffers[0].data + offset** points to the sync word of the next frame to be decoded. A successful execution of **tmalAdecAc3FindSyncword** guarantees this.

Figure 11 shows how the synchronization between the incoming data stream and the AC-3 decoder library functions must be implemented in non-streaming (push) mode. Note that the programmer must ensure that the AC-3 frame struct is set up properly before calling the core decoder function **tmalAdecAc3DecodeFrame**.

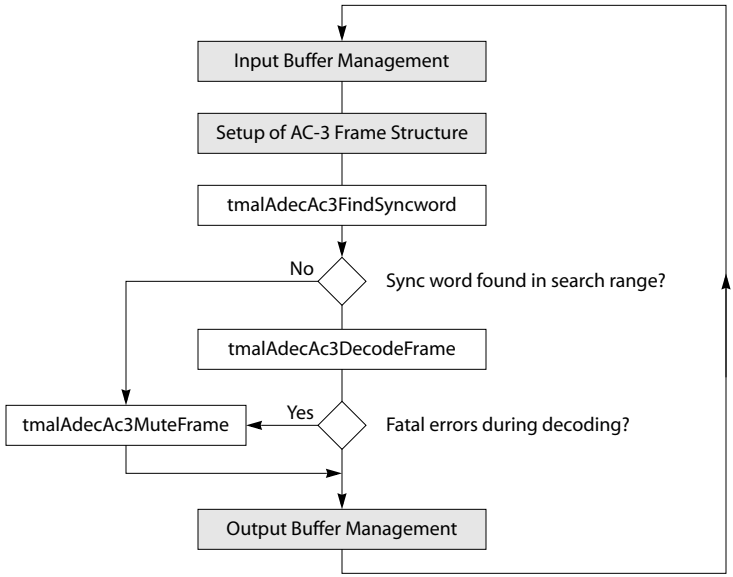


Figure 11 Mode of operation of the decoder using the AL library in push (non-streaming) mode. Shaded blocks represent functions to be implemented by the programmer. White blocks stand for functions provided by the AC-3 decoder library.

Obviously the delay of a decoder built up with the AL library in push mode adds up from the time required to get the input frame and the time required to decode the frame. This is acceptable for applications that do the video/audio synchronization as a separate task on the TriMedia, or when the AC-3 data is delivered with meaningful time stamps.

In applications transmitting AC-3 data without any synchronization information, it is important to keep the decoder delay in a certain range known to the rest of the audio/video system. The maximum delays are well-defined for different application environments. Dolby defines the requirements in the document *Dolby AC-3 Multichannel Digital Audio Decoding System For Consumer Products, Version 1* as of October 10, 1995.

To keep the overall decoding delay low in non-streaming (push) mode, it is important to implement a suitable input buffer management. The input buffer should be implemented as a relatively small FIFO.

Applications using the TriMedia AC-3 decoder library in streaming mode (either in OL or AL mode) can keep the decoding delay as low as possible since the core decoder function

implemented in **tmalAdecAc3Start** works block-oriented. The delay is determined by the granularity of the incoming AC-3 packets.

The larger the packets, the longer the decoder delay will be. At its input, the decoder accepts any packet size. The tradeoff is that with smaller packets, communication and internal synchronization overhead increases. The library would consume more CPU cycles.

illustrates the arbitrary size of the output PCM packets. Since the granularity of the decoder output is a multiple of 256 samples times the chosen number of channels, set the output packet buffer size accordingly. If you choose a smaller buffer size, the decoder sends multiple packets after decoding of one AC-3 audio block. The last of those packets may not be filled completely. This is possible since the data size of an audio packet may be smaller than its buffer size. If the output packet buffer size is larger than 256 times the number of channels, the decoder sends (via the dataout function) a complete PCM samples block when decoded. The remainder of the packet buffer is empty in this case.

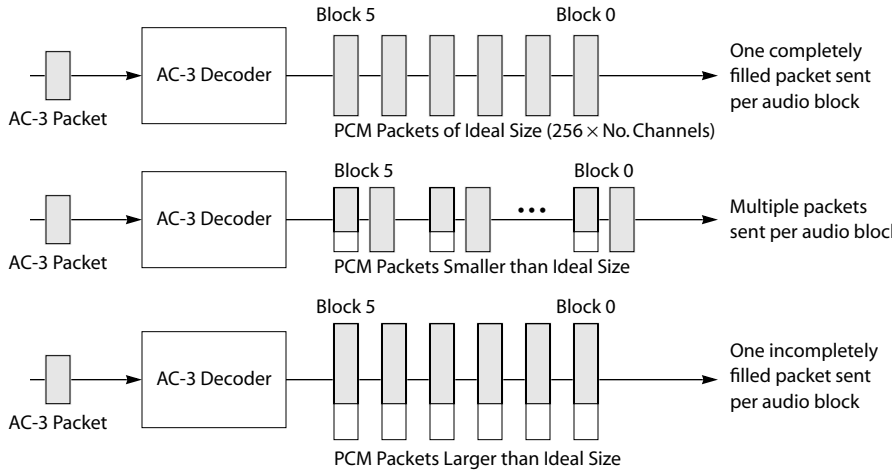


Figure 12 Fullness of PCM packets.

Dolby's latency requirements can be satisfied by TriMedia-based implementations, even though some overhead in the reading of the AC-3 data exists compared to specialized DSP solutions. However, the time being lost in the data-input processing can be compensated by the decoder core because this is running 3 to 5 times faster than required. The exact speed of the decoder does not need to be known exactly, since the synchronization between input and output will be done on time-stamp basis. An audio input task provides the incoming S/P DIF samples with time stamps, which can be interpreted as presentation time stamps. Therefore, the Dolby latency tables must be known to this task. This task also provides all other involved components with a reference clock. The AC-3 decoder then decodes the data and provides the output with the same time stamps. Finally the audio renderer ensures that the decoded sound is displayed at the correct

time with respect to the presentation time stamps. Note that the current version of the audio renderer does not support interpretation of presentation time stamps.

The current AC-3 library is fully functional for time-correct applications, since all synchronization issues are to be implemented in components external to the AC-3 decoder library.

Time Stamps

When used in streaming mode, the AC-3 decoder library is capable of the proper handling of time stamps. All incoming time stamps are interpreted as presentation time stamps (PTS). Decoding time stamps are not supported by the Ac-3 decoder library! The decoder selects a time stamp from the incoming AC-3 packets and propagates this to the outgoing PCM packets. If a valid time stamp is received it is assigned to the first PCM samples packet of the next decoded AC-3 frame. If the input packet, however, contains the beginning of an AC-3 frame and a valid time stamp, the time stamp is used for the first PCM block of the present frame.

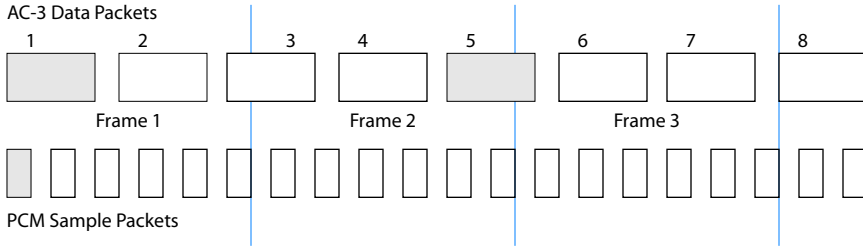


Figure 13 Assignment of Time Stamps

Figure 13 illustrates the time stamp assignment algorithm. The shaded packets represent packets with valid time stamps. The first AC-3 data packet contains a valid time stamp and it coincides with a frame start. It is therefore assigned to the first PCM packet. AC-3 packet number five contains the next valid time stamp. Since it is not aligned with a frame start it is assigned to the first PCM packet of the next audio frame. This algorithm is described in the ATSC standard.

Time Stamps for the Secondary Stereo Output

The handling of time stamps for the second output pin of the AC-3 decoder library depends on the mode of operation. If the pin is used to provide a stereo (ProLogic) downmix of the decoded audio signal, the PCM data packets receive the same time stamps as those of the main multichannel output. If the output is, however, configured to send IEC61937 formatted AC-3 packets, the time stamps need to be corrected. The decoding delay of an external decoder must be taken into account in order to maintain the synchronization between audio and video. Accordingly, the AC-3 decoder calculates the delay based on the sampling frequency and the data rate of the AC-3 bit stream and

subtracts it from the presentation time stamps attached to the IEC61937 data packets. The time stamp correction requires the knowledge of the clock reference. It is only applied when a clock handle is installed during the setup of the AC-3 decoder. Otherwise, the time stamps for the 1937 packets stay unmodified.

Rejection of Expired Input Packets

In addition to the assignment of time stamps, the AC-3 decoder is also capable of rejecting input packets with expired time stamps. This functionality can be enabled by assigning a clock handle during the setup of the AC-3 decoder. The clock handle is a field of the default instance setup structure which also contains the callback function pointers. When enabled, the rejection algorithm compares the time stamp to be assigned to the next output packet with the current system time which is determined with the clock handle. In an ATSC system, the system time is derived from the program clock reference. If the value of the PTS equals the system time or is even greater, the current input packet is marked as empty is send back on the empty input queue. No output data is produced for the associated AC-3 frame. Then, the AC-3 decoder searches for new synch word to continue decoding. Whenever an input packet is rejected the decoder's error callback function is called with the error message **A3_ERR_PTS_EXPIRED**.

Memory Allocation

All memory allocation for the AC-3 decoder is done during the configuration phase. Operated at the OL layer, the decoder allocates all required dynamic memory in the function **tmolAdecAc3Open** using **calloc**. During the processing no further memory allocation is required. Therefore, no callback function for memory allocation or freeing need to be installed. The dynamically allocated memory is freed by **tmolAdecAc3Close**.

When the AC-3 decoder library is used at the AL layer in either streaming or non-streaming mode dynamic memory is allocated by the functions **tmalAdecAc3Open** and **tmalAdecAc3GetInstanceSetup**. It is freed by **tmalAdecAc3Close**. As at the OL layer the memory gets allocated by **calloc**.

Callback Function Requirements

This section describes the functionality implemented in the function **tmalAdecAc3Start**. It is necessary for an application programmer to know under which conditions callback functions are called, and what actions the library expects to be carried out by the callback functions. Figure 14, following, illustrates how **tmalAdecAc3Start** works.

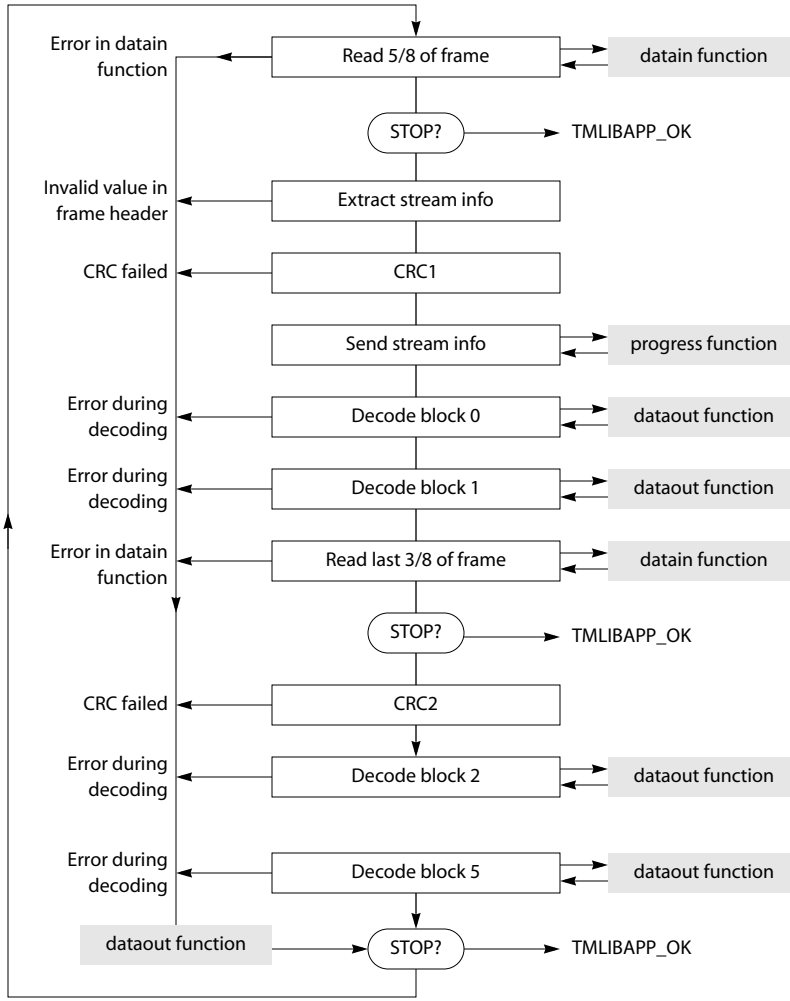


Figure 14 Flow diagram of **tmalAdecAc3Start**. Gray boxes without black lined border represent the callback functions called by the AL library. All other boxes stand for subfunctions implemented in **tmalAdecAc3Start**.

At first, the function **tmalAdecAc3Start** performs a check to see if the setup has been done and the instance ID is valid. If one of the checks fails, the function returns to its caller with the error message **TMLIBAPP_ERR_INVALID_INSTANCE** or **TMLIBAPP_ERR_NOT_SETUP**, respectively.

The decoder then goes into loop mode which it leaves only if either **tmolAdecAc3Stop** or **tmalAdecAc3Stop** has been called. There are two ways to exit the start loop. Operated in pull mode at the AL layer, a flag is set by **tmalAdecAc3Stop**. This stop flag is checked at

three locations within the start function—after receiving the first and second part of the frame data, and at the end of the loop. In OL layer mode, the return values of the datain and dataout callback functions are checked. If a return value is **TMLIBAPP_STOP_REQUESTED**, the decoder also leaves its main processing loop. This is part of the OL layer default stop mechanism and happens after **tmolAdecAc3Stop** has been called.

As its first task in the loop, the decoder tries to fill the internal decoding buffer up to the level that equals 5/8 of the current frame. This is the amount of data required to do the first CRC calculation. If not enough data is available, the datain callback function gets called. Dependent on the granularity of the input data packets, the datain function may get called several times. By returning a value unequal to **TMLIBAPP_OK**, the datain function indicates that an error occurred during the reception of input data packets. The AC-3 decoder then mutes the complete frame. It, therefore, calls the dataout function as often as necessary to send six PCM blocks containing 256 PCM samples across all channels of the respective format of the output pin (see also Frame versus Block-Oriented Decoding on page 26, which describes the output behavior of the decoder). After generating and sending of the PCM frame, the error callback function is called and gets as an argument the error code previously received from datain function. Program execution then continues at the end of the loop.

While searching for the first valid frame, the AC-3 decoder omits the muting.

All rectangular blocks of Figure 14 implement identical error handling. When an error occurs in one of those blocks, muted PCM data is generated for the not yet decoded audio blocks and sent by calling the dataout callback function. After that, the error callback function gets called. These callback function calls are not indicated in . It is not required for a user of the API to implement special error processing. The decoder core itself cares for producing the required amount of PCM data by muting or repeating of audio blocks. It is possible to count the occurrences of certain error types in the error callback function and to stop the decoder by calling of **tmalAdecAc3Stop**. This might be useful if the input delivers only corrupted frames for a certain time period.

After successfully reading of the first part of the frame, the decoder checks whether the stop function was called. If it was, the decoder calls the datain function to return the currently used packet as empty. This is required if queue-based data-transfer mechanisms are used in the datain callback function. After that, **tmalAdecAc3Start** is left with the return value **TMLIBAPP_OK**.

If the decoder continues, the next step is the extraction of the bitstream header field. The bitstream information is used internally by the decoder and can be passed to the application by the progress callback function. Since the internal checking of its validity is not reliable, the CRC is calculated before this information is passed to the application.

If the first CRC test fails, six muted audio blocks are produced in the fashion described previously, the error callback function is called with the error value **A3_ERR_CRC1_FAILED,** and, finally, the program continues at the stop block at the bottom of Figure 14.

The bitstream information is then passed to the application via the progress callback function. The user of the API determines how often this information shall be sent with the progress flags **A3_PROG_REPORT_FORMAT** , **A3_PROG_REPORT_CHANGES,** and **A3_PROG_REPORT_EVERY_FRAME**. Refer to section *AdecAc3 Progress* on page 45 for more information about how to use these progress flags.

After positive CRC test, the first audio blocks are decoded. The decoder sends the PCM packets on a block basis to the dataout callback function.

Now, the second part of the frame is decoded. The applied processing is the same as for the first part, with the difference that four audio blocks are decoded instead of two, and no further bitstream information is extracted. Finally, a check is made to see whether the stop function was called. If not, the processing of the next frame is started.

The current version of the AC-3 decoder library uses only the datain, dataout, error, and progress callback functions at the AL layer. No dynamic memory allocation is performed within the main processing all. The required memory is allocated by the Open function.

Application Requirements and Limitations

This section discusses integration issues, including what components may be connected to the TriMedia AC-3 library. Also discussed are what features must be implemented outside of the TriMedia library to implement a consumer product. Most of the following considerations are taken from the document *Dolby AC-3 Multichannel Digital Audio Decoding System For Consumer Products, Version 1* as of October 10, 1995.

Input Processing

If one intends to use the TriMedia AC-3 decoder library for decoding AC-3 bitstreams coming from external sources, it is necessary to provide an S/P DIF interface for the data transmission. The TriMedia itself does not have such an audio interface, but it is possible to convert the incoming S/P DIF signals to I2S signals using the Philips TDA1315 integrated circuit. This chip provides a glueless interface to the TriMedia DSP and implements the interface conversion bidirectionally.

Output Processing Chain

In consumer products, some additional processing on the AC-3 decoder library output must be carried out, as shown in Figure 15. An important requirement is for instance the matching of the decoder output to the capabilities of the loud speakers connected to the decoder device. This processing is determined by the bass reproduction capabilities of the speakers and by their position in the room. Apart from mandatory signal-processing, additional features can be implemented. Some examples include virtual surround processing, reverberation processing to simulate theatre, hall, or stadium acoustics, and Pro Logic decoding.

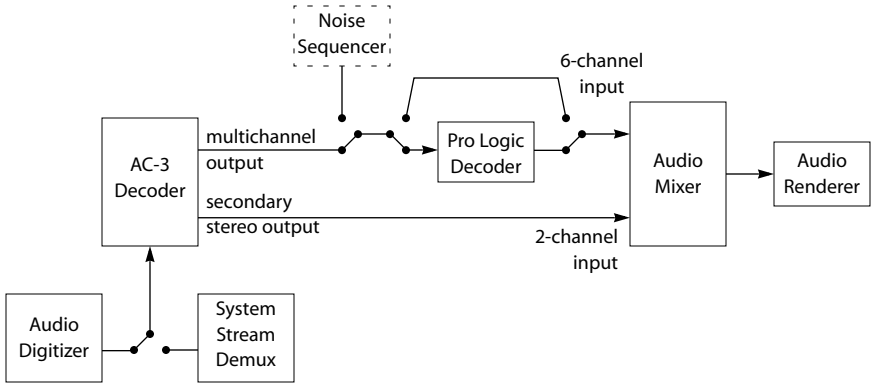


Figure 15 Output processing function blocks. The three blocks within the Mixer superblock are possible features of the TriMedia audio mixer library. The dashed Noise Sequencer block is not used during normal operation. It is only used for system calibration.

The first post-processing step on the AC-3 decoder output can be Pro Logic decoding. This makes sense in case the AC-3 bitstream contains only a stereo signal that was Pro Logic or Surround encoded prior to the AC-3 encoding. The Pro Logic decoder will be provided by Philips as a separate TriMedia library. The field **dSurMod** indicates this. It is returned by the progress callback function in the description field of the arguments. A Pro Logic decoder decodes Pro Logic encoded signals by applying an adaptive matrix to the two channels Lt and Rt. The result are four channels (L,C,R, and S). Optionally, the surround channel can be split into two surround channels that are attenuated by 3 dB.

Dolby distinguishes between two Dolby Surround (Pro Logic) application environments: home theatre and multimedia, PC-oriented solutions. The requirements are different with respect to the processing of the surround channel. A Dolby Surround Multimedia decoder skips the filters applied to the surround channel. The remainder of the processing is identical. See **http://www.dolby.com/multi/surreqrm.html** for further details. TriMedia's Pro Logic decoder library will support both application environments.

All audio environment specific processing is carried out in the Audio Mixer, which is also available in form of a TSSA library. Its task is it to adapt the output of the AC-3 or Pro Logic decoder to the audio reproduction environment. It performs adequate crossover/bass redirection filtering, which means that the bass content is distributed to the existing loudspeakers with respect to their bass reproduction capabilities. That basically means that channels connected to small speakers with limited bass capabilities are highpass-filtered. The low-frequency content is then distributed to speakers with better bass reproduction, or directly to the subwoofer. The second major task of the Audio Mixer is the delay of the surround and (for special listening environments) the center channel. The reason for channel delays is that it is desirable to have the sound of the five main speakers arriving simultaneously. For AC-3 decoding applications, a surround delay of up to 15 ms is required. For Pro Logic decoded signals, up to 30 ms is required, depending on the

relative position of the listener to front and surround speakers. the Audio Mixer also implements trims for the individual channels and volume control. Another important feature is that the mixer is capable of dealing with the secondary stereo output of the AC-3 decoder. It is interleaving the channels, so that they be rendered accurately by the audio renderer.

For headphone or stereo speaker environments, another special processing step might be applied. This is known as **stereo enhancement**, **virtual surround**, or **3-D sound processing**. Its purpose is to produce a virtual 3-D positioning effect by mixing the available channels down to two, and taking into account the head-related transfer function (HRTF). This approach provides good results when the position of the listener is known in advance, or when the employed processing is adapted to the approach. It doesn't work well for a wide hearing field.

Apart from HRTF processing, reverberation is a processing step that might be performed in the audio system. This processing block would be executed after the Audio Mixer.

System Calibration

As a further component for both AC-3 and Pro Logic decoding applications, a test signal generator is required. The calibration of the system can be done by using a pink noise sequencer. Philips provides such a function block that will produce PCM packets. The output of this function block can be connected to the mixer input.

Quality Assurance and Decoder Performance

This section describes how the quality and functionality of the AC-3 decoder library is assured. Furthermore, the library performance is also analyzed.

Quality Assurance

The quality of the AC-3 library is assured by design of the software and an extensive test suite. The decoder design was based upon the reference C code version 3.11 available from Dolby. During the entire implementation and optimization phase, extensive compliance tests were performed. During the implementation phase, a set of more than 400 test vectors covering almost all possible decoder states was used to assure compliance. The tests were carried out with an automatic test environment generating reference PCM outputs using the Dolby reference decoder, running with double-precision floating-point arithmetic. These reference PCM outputs were compared against the output of the TriMedia AC-3 decoder. The comparison was done on sample basis in time domain. The acceptance criterion was the highest absolute difference between the original samples and the TriMedia output, which in all cases was at most one least-significant bit in 16- and 18-bit mode. In 20-bit mode, some 2-bit differences occur.

In addition to this testing in the time-domain, intensive tests were carried out in the frequency domain, using Dolby's audio precision 2 test suite. No differences to the reference results have been observed.

Apart from the testing of the implemented algorithm comprehensive tests of the TSSA interface are performed to guarantee that the library is capable of dealing with all specified packet types.

Dolby Laboratories also evaluated the library core. According to Dolby's test results, the implemented algorithm is compliant with their class A product specification, which means that the output precision is as high as 20-bit.

Furthermore, extensive listening tests were carried out with different applications built upon the Dolby Digital AC-3 library.

Decoder Performance

This section provides an overview of the processor resources requirements for decoding AC-3 streams under different conditions. During the tests, the processor load of the decoder core was measured using two different test programs that were originally developed to perform the library compliance test in streaming and non-streaming mode, and the OL layer example **exolAdecAc3**. While the two compliance test programs have a very flexible interface capable of testing all possible decoder configurations, **exolAdecAc3** is relatively simple and therefore only a subset of the test modes were carried out with it. All three programs measure the decoder performance using the TriMedia custom operators **CYCLES** and **HICYCLES**. To provide a processor speed independent value the results are given in MIPS, assuming that a 100 MHz TriMedia processor accommodates 100 MIPS. All performance tests were executed on real TriMedia chips as opposed to simulations.

During the tests with the compliance test programs only the performance of the decoder core function was measured. This is **tmalAdecAc3DecodeFrame** in non-streaming mode and **tmalAdecAc3Start** in streaming mode. It is obvious that the non-streaming mode results are lower than those of the streaming mode because the required input data buffer management is implemented by the application in non-streaming mode. It therefore contributes only to the streaming mode results because **tmalAdecAc3Start** implements it in a hidden way. The results of the AL layer streaming mode tests do not include time spent in the callback functions.

In a real multitasking application using the AC-3 library at the OL layer, a certain overhead is added due to task switches and the time spent in the default data input and output functions. To provide an idea how big this overhead is, some test results are provided which represent the amount of MIPS actually spent in the AC-3 task.

The measurements were performed on seven different AC-3 streams with different sampling rates, data rates, and channel configurations. The example bitstreams *5voices*, *cave*, *locomotive,* and *egypt* are demo material available from Dolby. They can be sampled from laser disk or DVD. The bitstreams *wrst3841, music3* and *music4* originate from a Dolby

test vector suite; *wrst3841* is supposed to have worst-case characteristics with respect to the load of a decoder.

Table 1 Properties of the Test Streams

Test Stream	Sampling Rate	Data Rate	Channel Config
5voices	48 kHz	384 kbps	5.1
cave	48 kHz	448 kbps	5.1
egypt	48 kHz	448 kbps	5.1
locomotive	48 kHz	448 kbps	5.1
wrst3841	48 kHz	384 kbps	5.1
music3	48 kHz	384 kbps	5.0
music4	44.1 kHz	192 kbps	2.0

Both AL layer tests were carried out with four different decoder configurations. The OL layer test were performed with the first, second, and the fifth test configuration. The **Packet Type** column describes the PCM packet format used to store the decoded samples. In 32 bit mode the output sample precision was set to 20 bits. The **Downmix to** column contains the applied downmix operations for the test configurations and LFE Decoding tells whether or not the LFE channel in the input bitstream, if present, is decoded during the respective test. The 2nd Output column indicates is the stereo output of the decoder is used and, if so, if it carries compressed AC-3 data in 1937 format or stereo PCM data.

Table 2 Test Configurations

Test	Packet Type	Downmix to	LFE Decoding	2nd Output
Config1	apfFiveDotOne16	no Downmix	On	not used
Config2	apfFiveDotOne16	stereo, Pro Logic encoded	On	not used
Config3	apfFiveDotOne32	no Downmix	On	not used
Config4	apfStereo32	stereo, Pro Logic encoded	Off	not used
Config5	apfFiveDotOne16	no Downmix	On	atf1937

Table 3 Results of Non-Streaming Mode Test

Test Stream	Config 1	Config 2	Config 3	Config 4	Config 5
5voices	23.16 MIPS	20.95 MIPS	not tested	not tested	not tested
cave	23.54 MIPS	21.28 MIPS	not tested	not tested	not tested
egypt	23.30 MIPS	21.03 MIPS	not tested	not tested	not tested
locomotive	23.29 MIPS	21.07 MIPS	not tested	not tested	not tested

Table 3 Results of Non-Streaming Mode Test

Test Stream	Config 1	Config 2	Config 3	Config 4	Config 5
wrst3841	24.88 MIPS	22.57 MIPS	25.34 MIPS	21.31 MIPS	not tested
music3	24.04 MIPS	20.07 MIPS	not tested	not tested	not tested
music4	13.97 MIPS	not tested	not tested	not tested	not tested

Table 4 Results of AL Layer Streaming Mode Test

Test Stream	Config 1	Config 2	Config 3	Config 4	Config 5
5voices	24.01 MIPS	21.84 MIPS	not tested	not tested	not tested
cave	24.58 MIPS	22.26 MIPS	not tested	not tested	not tested
egypt	24.32 MIPS	22.08 MIPS	not tested	not tested	not tested
locomotive	24.27 MIPS	22.03 MIPS	not tested	not tested	not tested
wrst3841	25.70 MIPS	23.47 MIPS	26.16 MIPS	22.38 MIPS	not tested
music3	25.13 MIPS	21.74 MIPS	not tested	not tested	not tested
music4	14.81 MIPS	not tested	not tested	not tested	not tested

Table 5 Results of OL Layer Streaming Mode Test

Test Stream	Config 1	Config 2	Config 3	Config 4	Config 5
5voices	not tested	not tested	not tested	not tested	29.00 MIPS
cave	not tested	not tested	not tested	not tested	29.50 MIPS
egypt	not tested	not tested	not tested	not tested	29.30 MIPS
locomotive	not tested	not tested	not tested	not tested	29.30 MIPS
wrst3841	not tested	not tested	not tested	not tested	30.95 MIPS
music3	not tested	not tested	not tested	not tested	29.70 MIPS
music4	not tested	not tested	not tested	not tested	17.30 MIPS

The test results show clearly that a user of the AC-3 library must carefully decide whether to use the AL or OL layer interface and choose the appropriate packet type for the intended application. The size of the input and output packets also influences the decoder performance. If the processor load caused by the decoder is not important it is highly recommended to use the OL layer interface because of its ease of use.

AdecAc3 Inputs and Outputs

The TriMedia AC-3 decoder library provides one input pin. While the decoder being operated in streaming mode can handle both output pins, only one output is supported in non-streaming mode.

Inputs

The input pin of the TriMedia AC-3 decoder library is capable of handling two different AC-3 formats, **atfAC3** and **atf1937**. The difference between these formats is the way AC-3 frames are placed in the bitstream formed of consecutive data packets. Both formats are used for different application environments.

- atfAC3 bitstreams consist of pure AC-3 data in form of adjacent frames. The data rate is derived from the frame sizes and the sampling rate, which determines the number of frames per time period. There is no additional data present. This data format is typically used in MPEG systems. A system stream demultiplexer is the source of the AC-3 data in this case.
- atf1937 bitstreams consist of single AC-3 frames embedded in data blocks of 6144 bytes. The first 8 bytes of the data blocks form a preamble describing the type of data and the length of the block. Next to the preamble comes the actual AC-3 frame. The remainder of the data block is filled with zeros, the so called stuffing. This format is compliant to the digital audio interface format IEC958 which is also known as S/P DIF. The data rate is determined by the sampling frequency of the encoded AC-3 data. In 44.1 kHz mode it equals the data rate produced by an audio CD player. This data format is used in applications where the AC-3 data stream source resides in a system external to the TriMedia processor, for instance a laser disc player or a DVD player.

Main Multichannel Output

The first output pin of the AC-3 decoder carries PCM audio samples in various formats ranging from mono 16-bit to 6 channel 32-bit. It is also called “Main Multichannel Channel Output” earlier in this document.

It is possible to map the encoded audio content of any AC-3 bitstream, which ranges from 1 to 5 full bandwidth channels plus an optional subwoofer channel, to any number of existing output channels. The configuration parameters **outputMode** and **outLfeOn** determine the output configuration as described in depth on 1 and 1. It must be ensured by the application that the format specified for the output descriptor of this output pin matches the output configuration settings of the decoder. This means more specifically that no channels may be produced by the decoder that are not present in the output packet format.

The following table shows which **outputMode** and **outLfeOn** settings are supported dependent on the selected packet type. If the chosen **outputMode** does not match the

used packet type, the error message **A3_ERR_OUTPUT_MISMATCH** is returned by **tmalAdecAc3InstanceSetup**, **tmolAdecAc3InstanceSetup**, **tmalAdecAc3InstanceConfig**, and **tmolAdecAc3InstanceConfig**.

Table 6 Supported output formats for the first output

Packet Type	Allowed Values for outputMode and outLfeOn									Interleaved Channel Order at Output
	0	1	2	3	4	5	6	7	Lfe	
apfMono16		•								C
apfStereo16	•		•							L, R
apfFourCh_3_1_0_16	•	•	•	•	•	•				L, R, C, S
apfFourCh_2_2_0_16	•		•		•		•			L, R, ls, rs
apfFourCh_2_1_1_16	•		•		•				•	L, R, S, Lfe
apfFourCh_3_0_1_16	•	•	•	•					•	L, R, C, Lfe
apfFiveDotOne16	•	•	•	•	•	•	•	•	•	L, R, C, Lfe, ls, rs
apfMono32		•								C
apfStereo32	•		•							L, R
apfFourCh_3_1_0_32	•	•	•	•	•	•				L, R, C, S
apfFourCh_2_2_0_32	•		•		•		•			L, R, ls, rs
apfFourCh_2_1_1_32	•		•		•				•	L, R, S, Lfe
apfFourCh_3_0_1_32	•	•	•	•					•	L, R, C, Lfe
apfFiveDotOne32	•	•	•	•	•	•	•	•	•	L, R, C, Lfe, ls, rs

The meaning of the channel abbreviations is as follows:

L = left
R = right
C = center
Lfe = subwoofer
S = surround (if only one surround channel is used; this channel is mapped to ls)
ls = left surround
rs = right surround

Enums are defined for both the values of **outputMode** and **outLfeOn**, refer to **tmAdecAc3-OutConfig_t** and **tmAdecAc3LfeMod_t**.

Channels that are supported by the packet format but not generated by the decoder running in a particular output mode (**outputMode** and **outLfeOn**) are filled with zeros.

The AC-3 decoder provides two different output sample resolutions if one of the 32-bit packet formats is chosen. It supports 18-bit and 20-bit precision. The precision is determined by the lower 8 bits in the description field of the first's output descriptor's format.

A bitmask **AVFORMAT_NUMBER_OF_BITS_MASK** to access this value is provided in *tmAvFormats.h*. The masked value tells the number of significant bits within the 32 bit samples. The bits are stored right justified.

The format of the first output is normally not installed prior to the start of the decoder because the sampling frequency is unknown and must be determined from information stored in the incoming AC-3 bitstream. An application can specify during the setup phase of the decoder what PCM format and precision the decoder should install later on with the **pcmFormatOut0** and **precisionOut0** fields of the instance setup variable. The default format is **apfFiveDotOne16**.

Secondary Stereo Output

The second output pin is optional and it is supported only in streaming mode. It is activated during instance setup if its output descriptor pointer is different from **NULL**. As its name "Secondary Stereo Output," used earlier in this document, leads to assume, it carries audio data packets in stereo format. The task of this output is to provide either a surround compatible downmixed version of the main audio output or AC-3 data encoded in IEC958/1937 format.

As for the first output pin, the format for the second output can be selected in two different ways:

- the format information of the output descriptor is used,
- the information stored in **formatOut1, precisionOut1, stereoMixMode** and **dualMonoMode1** is used.

If the output descriptor for the second output contains a format pointer different from **NULL**, the decoder takes the respective information from this format structure. In this case the **dataType** field determines whether the 1937 format or linear PCM is used. When linear PCM is selected, the **dataSubtype** field must have either the value **apfStereo16** or **apfStereo32**. In 32-bit mode the least significant 8 bits of the description field determine the output precision of the decoder. Supported values are 18- and 20-bit. It is determined by the ProLogic bit in the description field of the format whether a ProLogic downmix is performed or a regular stereo downmix. The respective bit position in the description field is predefined as **AVFORMAT_PROLOGIC_ENCODED**.

If no format is present in the output descriptor when **tmXlAdecAc3InstanceSetup** is called, the configuration of the output pin is performed based on the settings of the **formatOut1** and **precisionOut1** fields of the instance setup variable; **formatOut1** determines if the output carries compressed AC-3 data (**atf1937**) or stereo PCM data (**atfLinearPCM**). In the second case, the **precisionOut1** field determines the **dataSubtype** of the output format. This format will be installed by the decoder after the sampling frequency information is obtained from the AC-3 bitstream. If the precision field contains the value 16 the output format data subtype is set to **apfStereo16**, otherwise **apfStereo32**. The type of the stereo downmix is determined by the **stereoMixMode** field. The permitted values are: **A3_SECOND_OUTPUT_STEREO_MIX** and **A3_SECOND_OUTPUT_PROLOGIC_MIX**. In

addition to the stereo downmixing mode and the sample precision, the dual mono behavior of the second output can be determined with the instance setup field **dualMonoMode1**. For its meaning, see page 61.

The downmixed stereo data can be used for a headphone connector. It could also be fed to an external Dolby Surround receiver. This receiver is then capable of reproducing a surround audio field.

Note
The second output always works in line out mode irrespective of the compression mode setting for the main output!

Running in **atf1937** format, the decoder performs a formatting of the raw AC-3 data which is compatible with the S/P DIF format. This output can be used to connect to an external AC-3 decoder. A possible application scenario is a product with build in speakers and a digital audio output. If available, an external home theater system can be used instead of the internal speakers. There are also a couple of products on the market that provide only a stereo line output plus the digital output supplying the AC-3 data. The amount of data produced in 1937 mode is identical to the amount produced in **apfStereo16** mode. Six data packets with a **dataSize** of 1024 bytes are sent at this output pin for each decoded AC-3 frame. If the **bufSize** of the empty output packets is smaller than 1024 bytes, the decoder asserts in debug mode. The application must ensure that smaller packets are not used.

During normal operation the **atf1937** data packets are sent in a sequence after the second audio block of the AC-3 frame is decoded and the respective block of PCM samples is sent at the main multichannel output pin. If internal decoding errors occur and muting is applied, first all 1536 samples of the main multichannel audio output are sent at the first output pin and then the AC-3 data is sent at the second output pin.

AdecAc3 Errors

This section describes all errors that can be reported by the error callback function. The error callback function is called from the AL layer start function which implements the actual AC-3 decoding in streaming mode. Only the first three errors cause the decoder to return from the decoding task. The other errors are non-fatal, the decoder continues to work after execution of the error callback function.

`TMLIBAPP_ERR_INVALID_INSTANCE`	AL instance is invalid.
`TMLIBAPP_ERR_NOT_SETUP`	Decoder is not yet set up.
`TMLIBAPP_ERR_ALREADY_STARTED`	The instance of the decoder is already started.
`A3_ERR_SYNC_NOT_FOUND`	Decoder did not find a valid sync word within the search range. The search range is the width of last correctly decoded frame plus two bytes, or, if the input format is **atf1937**, 6144 bytes. To maintain a continuous data stream at the output the decoder mutes one frame (1536 samples).
`A3_ERR_DECODE_FATAL`	A fatal error occurred during the decoding of the last block. The decoder performs muting of the current audio block and for the rest of the frame after returning from the error callback function.
`A3_ERR_CRC1_FAILED`	The calculation of the first CRC derived a wrong value. Depending on how many blocks have already been repeated, block repeats or muting is performed after the return from the error callback function. The maximum number of repeated blocks can be determined with the **maxRepeat** field in the instance setup structure.
`A3_ERR_CRC2_FAILED`	The calculation of the second CRC yielded a wrong value. Therefore, for the last four blocks of the current frame block repeats or muting is performed after return from the error callback function.
`A3_ERR_PTS_EXPIRED`	The valid time stamp, that has been received last, is expired and the current input packet is therefore rejected. For more information, refer to the section *Rejection of Expired Input Packets* on page 31.

Aside from the first three error messages, all messages have an informative nature. The application gets the information that something went wrong during the search for a new sync word or during the actual decoding of AC-3 data. It is taken care of appropriate error handling by the decoder. The application should implement special processing in the case that no sync word could be found within a certain amount of input data. In this case something goes wrong upstream.

AdecAc3 Progress

The purpose of the progress function is to provide the application with information on the properties of the AC-3 bitstream. This is required to properly set up the audio reproduction system and display relevant information. A user of the TriMedia AC-3 decoder library can select between getting the bitstream information only once with the reception of the first AC-3 frame, for every frame, or only when important changes of the configuration occur. This choice is made by setting the **progressReportFlags** in the **defaultSetup** to one of the following values. Refer to section *Setup of an OL Layer Decoder Application* on page 20 for an example on how to configure the progress function.

Table 7 Supported Progress Flags

progressReportFlags	meaning
A3_PROG_REPORT_FORMAT	Progress function reports bitstream format only for the first frame and then never again.
A3_PROG_REPORT_EVERY_FRAME	Progress function reports bitstream format for every frame.
A3_PROG_REPORT_CHANGES	Progress function reports bitstream format only when either **acMod**, **lfeOn**, **sFrequency**, or **dataRate** changed.

If none of the above values is assigned to the **progressReportFlags** field, the progress report function will never be called by the decoder. If **A3_PROG_REPORT_EVERY_FRAME** or **A3_PROG_REPORT_CHANGES** is chosen **A3_PROG_REPORT_FORMAT** is implicitly activated. The progress function indicates to the application which flag has triggered its call via its second parameter. The time the progress function is called, the flag value is always **A3_PROG_REPORT_FORMAT**.

A pointer to a structure of the type **tmAdecAc3HeaderInfo_t** returns the bitstream information from the decoder to the application. The following code fragment shows how to access the sampling frequency information within the progress callback function.

```
tmLibappErr_t decoderProgFunc(
   Int instId, UInt32 flags, ptsaProgressArgs_t args
){
   ptmAdecAc3HeaderInfo_t config;
   Float32               freq;
   config = (ptmAdecAc3HeaderInfo_t) args->progressCode;
   freq   = config->sFrequency;
```

For more information on the fields of **tmAdecAc3HeaderInfo_t**, see page 69.

AdecAc3 Configuration

The TriMedia AC-3 decoder library provides two different functions to change the configuration during run-time. One is the AL layer function **tmalAdecAc3InstanceConfig** and the other one is the OL layer function **tmolAdecAc3InstanceConfig**. They accept an identical set of commands. The difference between them is that the OL layer function executes its commands delayed via an operation system control queue, as opposed to **tmalAdecAc3InstanceConfig** which changes the internal decoder configuration directly when it is called. In fact, the actual command execution takes place in **tmalAdecAc3InstanceConfig** in both cases, **tmolAdecAc3InstanceConfig** just implements the queue handler.

The control queue is checked by the decoder every time a data packet is received or sent but changes are only applied when the decoder starts to decode a new frame. A typical delay of a configuration change is therefore up to 48 ms.

There are three different kinds of commands. An application can change the internal decoder configuration, obtain information on the current values of internal configuration parameters, and retrieve values for default settings. These default settings are provided for parameters that depend on the formats of the input/output descriptors.

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.

The remaining two fields are not used by **tmalAdecAc3InstanceConfig**. 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 **tmolAdecAc3InstanceConfig** with the return value of **tmalAdecAc3InstanceConfig**. The application must check this value as well as the return value of **tmolAdecAc3InstanceConfig** which indicates problems with the control queue.

An OL layer application must provide a control descriptor during the initialization phase (before calling **tmolAdecAc3InstanceSetup**). The queues of this descriptor carry the com-

mands between the application and the AC-3 decoder component. Refer to page 20 for more information about the proper setup of the decoder.

Table 8 Commands to Change the Configuratio

Command	Supported Values for Parameter
A3_CONFIG_SET_KARAOKE_MODE	all values defined in **tmAdecAc3KaraokeMode_t**
A3_CONFIG_SET_COMP_MODE	all values defined in **tmAdecAc3CompMode_t**
A3_CONFIG_SET_SUBWOOFER_ON	no parameter required
A3_CONFIG_SET_SUBWOOFER_OFF	no parameter required
A3_CONFIG_SET_OUTPUT_MODE	all values defined in **tmAdecAc3OutConfig_t**
A3_CONFIG_SET_DUAL_MONO_MODE	all values defined in **tmAdecAc3DualMonoMode_t**
A3_CONFIG_SET_DYN_RNG_SCALE_HI	values of the range [0.0 , 1.0]
A3_CONFIG_SET_DYN_RNG_SCALE_LOW	values of the range [0.0 , 1.0]
A3_CONFIG_SET_PCM_SCALE_FACTOR	values of the range [0.0 , 1.0]

Table 9 Commands to get Settings of current Configuration

Command	Parameter needs cast to
A3_CONFIG_GET_KARAOKE_MODE	tmAdecAc3KaraokeMode_t
A3_CONFIG_GET_COMP_MODE	tmAdecAc3CompMode_t
A3_CONFIG_GET_SUBWOOFER_MODE	tmAdecAc3LfeMod_t
A3_CONFIG_GET_OUTPUT_MODE	tmAdecAc3OutConfig_t
A3_CONFIG_GET_DUAL_MONO_MODE	tmAdecAc3DualMonoMode_t
A3_CONFIG_GET_DYN_RNG_SCALE_HI	Float32
A3_CONFIG_GET_DYN_RNG_SCALE_LOW	Float32
A3_CONFIG_GET_PCM_SCALE_FACTOR	Float32

Table 10 Commands to get default Settings

Command	Parameter needs cast to
A3_CONFIG_GET_DEF_SUBWOOFER_MODE	tmAdecAc3LfeMod_t
A3_CONFIG_GET_DEF_OUTPUT_MODE	tmAdecAc3OutConfig_t

The commands of Table 10 are used to retrieve recommended settings for the output channel configuration dependent on the packet format of the output descriptor. This could help to avoid configuration conflicts. If for instance the output format is **apfFour-**

Ch_3_1_0_16 output mode **A3_OUTCHANCONFIG_2_2** cannot be used since it produces two surround channels. The packet format, however, supports only one surround channel. In this particular case the recommended output mode would be **A3_OUTCHANCONFIG_3_1** and the recommended LFE mode **A3_LFE_OFF**, since no subwoofer channel is supported by this packet format.

If a floating point value is to be sent to the decoder or to be received by the application, a special casting mechanism is required. The dynamic range compression cut scale factor, for instance, can be set to 0.8 with the following operations:

```
tsaControlArgs_t cargs;
Float32    fval = 0.8;
cargs.command   = A3_CONFIG_SET_DYN_RNG_SCALE_HI;
cargs.parameter = *((Pointer *) &fval);
tmolAdecAc3InstanceConfig( decInstance, tsaControlWait, &cargs);
```

If the application wants to obtain the current setting of this decoder parameter it must do the following:

```
tsaControlArgs_t cargs;
Float32          fval;
cargs.command   = A3_CONFIG_GET_DYN_RNG_SCALE_HI;
tmolAdecAc3InstanceConfig( decInstance, tsaControlWait, &cargs);
fval = *((Float32 *) &cargs.parameter);
```

This casting is required because otherwise an implicit cast to integer would be performed by the compiler.

For additional information refer to **tmAdecAc3ConfigTypes_t** on page 62, to **tmalAdecAc3InstanceConfig** on page 86, and to **tmolAdecAc3InstanceConfig** on page 88.

Note
If **tmolAdecAc3InstanceConfig** is called within the context of the AC-3 decoder, the command gets executed immediately. This is the case, when this function gets called from the AC-3 decoder progress function. The queue mechanism is only used when the function call happens in a separate task.

AC-3 API Data Structures

This section presents the AC-3 Decoder device library data structures.

tmalAdecAc3LibraryMode_t

```
typedef enum {
   A3_LIB_MODE_PUSH,
   A3_LIB_MODE_PULL
} tmalAdecAc3LibraryMode_t;
```

Description

The decoder can be run in push mode or in pull mode (only in the AL layer). These are legal values for the field **libraryMode** in **tmalAdecAc3InstanceSetup_t**. Push mode is also known as non-streaming mode. Pull mode is also known as streaming mode.

For more information, refer to structures **tmalAdecAc3InstanceSetup_t**, **tmalAdecAc3-InstanceSetup**, and to *The AL Layer* on page 16.

tmAdecAc3ProgressFlags_t

```
typedef enum {
   A3_PROG_REPORT_FORMAT        = 0x00000001,
   A3_PROG_REPORT_EVERY_FRAME   = 0x00000002,
   A3_PROG_REPORT_CHANGES       = 0x00000004
} tmAdecAc3ProgressFlags_t;
```

Description

These enums fulfill two purposes:

- they are used as the progress flags that an application can set in the instance setup structure to configure the decoder,
- they are used by the progress function to indicate which event caused its call. Whenever the progress function is called its flags parameter equals one of these enums.

To configure the decoder, these flags are assigned to the **progressReportFlags** in the **tsaDefaultInstanceSetup_t** structure. Refer to *Setup of an OL Layer Decoder Application* on page 20 for an example on how to configure the progress function.

When the **A3_PROG_REPORT_FORMAT** flag is set, the decoder calls the progress function once after it successfully decoded the frame information of the initial frame. The argument **((ptsaProgressArgs_t)arg)–>progressCode** is a pointer to a struct of type **tmAdecAc3HeaderInfo_t**. It contains the header information of the frame to be decoded.

It is normally sufficient to propagate the bitstream information to an application only when important changes occur. When the flag **A3_PROG_REPORT_CHANGES** is set, the decoder calls the progress function when the first valid frame is found. After that, the progress function is called, only when changes in either the channel configuration, the data rate or the sampling rate occur. The progress function indicates the configuration change by using **A3_PROG_REPORT_CHANGES** as the flags parameter. However, at its first call the progress function is using **A3_PROG_REPORT_FORMAT** as the flags parameter.

When the **A3_PROG_REPORT_EVERY_FRAME** flag is set, the decoder calls the progress function every time it decodes the bitstream information of a valid new frame. Likewise for the flag **A3_PROG_REPORT_CHANGES**, the first call of the progress function is done with **A3_PROG_REPORT_FORMAT** as the flags parameter.

For more information, refer to structures **tmalAdecAc3InstanceSetup_t**, **tmalAdecAc3Start**, as well as the sections *Setup of an OL Layer Decoder Application* and *AdecAc3 Progress* earlier in this chapter.

tmAdecAc3AcMod_t

```
typedef enum {
   A3_ACMOD_DUAL_MONO  = 0x00000000,
   A3_ACMOD_1_0        = 0x00000001,
   A3_ACMOD_2_0        = 0x00000002,
   A3_ACMOD_3_0        = 0x00000003,
   A3_ACMOD_2_1        = 0x00000004,
   A3_ACMOD_3_1        = 0x00000005,
   A3_ACMOD_2_2        = 0x00000006,
   A3_ACMOD_3_2        = 0x00000007
} tmAdecAc3AcMod_t;
```

Description

These enum values are used to characterize the full bandwidth audio channel configuration of the AC-3 bitstream currently decoded. Enums of this type are used in the **acMod** field of **tmAdecAc3HeaderInfo_t** in conjunction with the progress callback function and the non-streaming mode decoder function **tmalAdecAc3DecodeFrame**.

The channel configuration is as follows:

Table 11 Audio Coding Modes of an AC-3 bitstream

Value	coded channels
A3_ACMOD_DUAL_MONO	(1+1) dual mono
A3_ACMOD_1_0	(1/0) mono
A3_ACMOD_2_0	(2/0) stereo
A3_ACMOD_3_0	(3/0) left, center, right
A3_ACMOD_2_1	(2/1) left, right, sur
A3_ACMOD_3_1	(3/1) left, center, right, sur
A3_ACMOD_2_2	(2/2) left, right, left sur, right sur
A3_ACMOD_3_2	(3/2) left, center, right, left sur, right sur

tmAdecAc3LfeMod_t

```
typedef enum {
   A3_LFE_OFF   = 0x00000000,
   A3_LFE_ON    = 0x00000001
} tmAdecAc3LfeMod_t;
```

Description

These enum values are used to detect if a subwoofer channel is encoded in the current AC-3 bitstream. An application can also use them to select whether or not the subwoofer channel, if present, shall be decoded.

The progress function returns the information whether a subwoofer channel is present in the current bitstream in the field **lfeOn** of the **tmAdecAc3HeaderInfo_t** struct. Refer also to *AdecAc3 Progress*.

Table 12 Subwoofer Modes in **tmAdecAc3HeaderInfo_t**

Value	Mode
A3_LFE_OFF	LFE channel does not exist in stream
A3_LFE_ON	LFE channel exists in stream

To configure the decoder either **A3_LFE_OFF** or **A3_LFE_ON** can be assigned to the **outLfeOn** field of **tmalAdecAc3InstanceConfig_t** before **tmalAdecAc3InstanceSetup** or **tmolAdecAc3Instance**, respectively, is called.

The instance config function also uses these enums as parameter values when the respective command concerns the subwoofer channel. (See page 46.)

Table 13 Subwoofer Modes for the Decoder Configuration

Value	Mode
A3_LFE_OFF	LFE channel is not decoded
A3_LFE_ON	LFE channel is decoded if it exists

tmAdecAc3SurMod_t

```
typedef enum {
   A3_SURMOD_NOT_INDICATED  = 0x00000000,
   A3_SURMOD_NOT_ENCODED    = 0x00000001,
   A3_SURMOD_ENCODED        = 0x00000002,
   A3_SURMOD_RESERVED       = 0x00000003
} tmAdecAc3SurMod_t;
```

Description

These enum values are used for stereo AC-3 bitstreams to determine whether or not the stereo signal is Dolby Surround encoded. In the case that the signal is Surround encoded, a Dolby Pro Logic decoder can be connected to the AC-3 output to produce a surround sound field. Enums of this type are used as values for the field **dSurMod** in **tmAdecAc3-HeaderInfo_t**.

Table 14 Dolby Surround Modes

Value	Mode
A3_SURMOD_NOT_INDICATED	unknown if audio is Surround encoded or not
A3_SURMOD_NOT_ENCODED	audio is not encoded
A3_SURMOD_ENCODED	audio is Surround encoded
A3_SURMOD_RESERVED	reserved value

tmAdecAc3RoomType_t

```
typedef enum {
   A3_ROOMTYPE_NOT_INDICATED    = 0x00000000,
   A3_ROOMTYPE_LARGE            = 0x00000001,
   A3_ROOMTYPE_SMALL            = 0x00000002,
   A3_ROOMTYPE_RESERVED         = 0x00000003
} tmAdecAc3RoomType_t;
```

Description

Some AC-3 bitstreams contain information on the properties of the recording environment. The fields **roomTyp** and **roomTyp2** (in 1+1 mode) of **tmAdecAc3HeaderInfo_t** are using the values defined by **tmAdecAc3LfeMod_t**. These values are meaningful only when **audProdIE** and **audProdi2E**, respectively, equal one.

In streaming mode this information can be obtained via the progress callback function, and in non-streaming via **tmalAdecAc3DecodeFrame**.

Table 15 Room Type Modes

Value	Mode
A3_ROOMTYPE_NOT_INDICATED	Room type is not indicated
A3_ROOMTYPE_LARGE	Production room with X curve monitor
A3_ROOMTYPE_SMALL	Production room with flat monitor
A3_ROOMTYPE_RESERVED	Reserved value

tmAdecAc3CopyRight_t

```
typedef enum {
   A3_COPYRIGHT_NOT_PROTECTED   = 0x00000000,
   A3_COPYRIGHT_PROTECTED       = 0x00000001
} tmAdecAc3CopyRight_t;
```

Description

These are the possible values of the **copyrightb** field of the struct **tmAdecAc3HeaderInfo_t**. They determine whether or not the audio content of the AC-3 bitstream is protected by copyright.

In streaming mode this information can be obtained via the progress callback function, and in non-streaming via **tmalAdecAc3DecodeFrame**.

tmAdecAc3CopyState_t

```
typedef enum {
   A3_COPYSTATE_COPY   = 0x00000000,
   A3_COPYSTATE_ORG    = 0x00000001
} tmAdecAc3CopyState_t;
```

Description

These are the possible values of the **origbs** field of the struct **tmAdecAc3HeaderInfo_t**. They determine whether or not the audio content of the AC-3 bitstream is an original bitstream or a copy of another bitstream.

In streaming mode this information can be obtained via the progress callback function, and in non-streaming via **tmalAdecAc3DecodeFrame**.

tmAdecAc3StereoOutputMixMode_t

```
typedef enum {
   A3_SECOND_OUTPPUT_STEREO_MIX    = 0x00000000,
   A3_SECOND_OUTPUT_PROLOGIC_MIX   = 0x00000001
} tmAdecAc3CopyState_t;
```

Description

These are the possible values of the **stereoMixMode** field of the structs **tmalAdecAc3InstanceSetup_t** and **tmolAdecAc3InstanceSetup_t**. They determine whether a regular stereo or a Dolby ProLogic Surround compatible downmix is performed on the second output when active.

tmAdecAc3OutConfig_t

```
typedef enum {
   A3_OUTCHANCONFIG_SUR    = 0x00000000,
   A3_OUTCHANCONFIG_1_0    = 0x00000001,
   A3_OUTCHANCONFIG_2_0    = 0x00000002,
   A3_OUTCHANCONFIG_3_0    = 0x00000003,
   A3_OUTCHANCONFIG_2_1    = 0x00000004,
   A3_OUTCHANCONFIG_3_1    = 0x00000005,
   A3_OUTCHANCONFIG_2_2    = 0x00000006,
   A3_OUTCHANCONFIG_3_2    = 0x00000007
} tmAdecAc3OutConfig_t;
```

Description

These enum values are used to select how many full bandwidth output channels are generated by the decoder. If the bitstream contains a channel configuration which does not match the selected output configuration downmixing is applied. Enums of this type are used in the **outputMode** field of **tmalAdecAc3InstanceConfig_t** to configure the decoder during the setup phase. During run-time the output configuration of the decoder can be re-configured by calling either **tmalAdecAc3InstanceConfig** or **tmolAdecAc3InstanceConfig** with the command **A3_CONFIG_SET_OUTPUT_MODE**. Information on the current output channel configuration and an optimal output configuration depending on the output descriptor format can be obtained with the commands **A3_CONFIG_GET_OUTPUT_MODE** and **A3_CONFIG_GET_DEF_OUTPUT_MODE**.

The channel configuration is as follows:

Table 16 Audio Coding Modes of an AC-3 bitstream

Value	Coded Channels
A3_OUTCHANCONFIG_SUR	(2/0) stereo Surround compatible encoded
A3_OUTCHANCONFIG_1_0	(1/0) mono
A3_OUTCHANCONFIG_2_0	(2/0) stereo
A3_OUTCHANCONFIG_3_0	(3/0) left, center, right
A3_OUTCHANCONFIG_2_1	(2/1) left, right, sur
A3_OUTCHANCONFIG_3_1	(3/1) left, center, right, sur
A3_OUTCHANCONFIG_2_2	(2/2) left, right, left sur, right sur
A3_OUTCHANCONFIG_3_2	(3/2) left, center, right, left sur, right sur

tmAdecAc3CompMode_t

```
typedef enum {
   A3_DYNRNGMODE_ANALOG_DIALNORM    = 0x00000000,
   A3_DYNRNGMODE_DIGITAL_DIALNORM   = 0x00000001,
   A3_DYNRNGMODE_LINE_OUT           = 0x00000002,
   A3_DYNRNGMODE_RF_REMOD           = 0x00000003
} tmAdecAc3CompMode_t;
```

Description

These enum values are used to select the dynamic range of the decoded audio signal which is required to map the reproduced audio to the listening environment and speaker capabilities. The enum values can be assigned to the **compMode** field of the struct **tmalAdecAc3InstanceConfig_t** to configure the decoder in the setup phase in conjunction with the instance setup functions. During the actual decoding the instance config functions can be used to obtain the current compression mode from the decoder or change the mode using the commands **A3_CONFIG_SET_COMP_MODE** or **A3_CONFIG_GET_COMP_MODE**, respectively.

When you choose **A3_DYNRNGMODE_ANALOG_DIALNORM** or **A3_DYNRNGMODE_DIGITAL_DIALNORM**, the **dynrng** values of the 6 audio blocks taken from the AC-3 bitstream determine the dynamic range of the audio output. The decoder application can scale the dynamic range with the scale factors **dynRngScaleHi** and **dynRngScaleLow** stored in the instance config struct during setup. If those scale values equal zero no dynamic range compression is applied, and if they equal one the full dynamic range compression is applied. If downmixing is active the channels are additionally attenuated by 11 dB. The difference between the modes **A3_DYNRNGMODE_ANALOG_DIALNORM** and **A3_DYNRNGMODE_DIGITAL_DIALNORM** is the handling of the dialog normalization. In the analog mode, the normalization is not done by the decoder. It can be implemented external to the decoder. In contrast to that, the dialog normalization is carried out by the decoder in the second mode.

In **A3_DYNRNGMODE_LINE_OUT** mode, the dynamic range of the signal is also determined by the individual **dynrng** values of the 6 audio blocks. Again the scale factors **dynRngScaleHi** and **dynRngScaleLow** are used to scale the dynamic range compression factors. Dialogue normalization is not applied in line out mode.

In **A3_DYNRNGMODE_RF_REMOD** mode, the heavy compression value **compr** is used to limit the dynamic range of the audio signal. This mode is intended to ensure that certain peak levels are not exceeded. It is appropriate for listening environments where disturbance of other people is to be avoided. This mode is also applied to prevent overmodulation when RF modulators are used.

tmAdecAc3KaraokeMode_t

```
typedef enum {
   A3_KARAOKEMODE_NO_VOCAL      = 0x00000000,
   A3_KARAOKEMODE_LEFT_VOCAL    = 0x00000001,
   A3_KARAOKEMODE_RIGHT_VOCAL   = 0x00000002,
   A3_KARAOKEMODE_BOTH_VOCALS   = 0x00000003
} tmAdecAc3KaraokeMode_t;
```

Description

These enum values are used to select the method how Karaoke bitstreams are handled. Karaoke bitstreams contain up to 5 main audio channels: left and right, M which is a guide melody and V1 and optionally V2 as vocal tracks. The TriMedia AC-3 decoder is Karaoke capable, which means a user can choose between four different reproduction modes.

Karaoke bitstreams are reproduced only with the front speakers since they do not carry surround information. In two speaker mode M is reproduced as phantom center, otherwise on the real center channel. If **A3_KARAOKEMODE_NO_VOCAL** is selected, none of the vocal tracks is reproduced. If **A3_KARAOKEMODE_LEFT_VOCAL** or **A3_KARAOKEMODE_RIGHT_VOCAL** is set, V1 or V2, respectively, is mixed either into the phantom center in stereo mode or into the real center in three channel mode. In **A3_KARAOKEMODE_BOTH_VOCALS** mode, V1 is mixed into the left channel and V2 into the right channel.

The Karaoke mode can be set during the setup phase in the kCapableMode field of the struct **tmalAdecAc3InstanceConfig_t**. During run-time the Karaoke mode can be changed by calling the instance config functions with the command **A3_CONFIG_SET_KARAOKE_MODE**. The information on what mode is currently set can be obtained with the command **A3_CONFIG_GET_KARAOKE_MODE**.

For more information on the Karaoke modes, see annex C of the ATSC document A/52.

tmAdecAc3DualMonoMode_t

```
typedef enum {
   A3_DUALMONOMODE_STEREO   = 0x00000000,
   A3_DUALMONOMODE_LEFT     = 0x00000001,
   A3_DUALMONOMODE_RIGHT    = 0x00000002,
   A3_DUALMONOMODE_MIXED    = 0x00000003
} tmAdecAc3DualMonoMode_t;
```

Description

These enum values are used to select the method how dual mono bitstreams are handled. In **A3_DUALMONOMODE_STEREO** mode the first mono channel is routed to the left output channel and the second mono channel to the right. In **A3_DUALMONOMODE_LEFT** mode the first mono channel is routed to the center output channel and in **A3_DUALMONOMODE_RIGHT** mode, it is the right channel, respectively. When **A3_DUALMONOMODE_MIXED** is set, the center channel will carry both mono channels attenuated by 3 dB and mixed together.

The dual mono reproduction mode can be set in the setup phase in the **dualMonoMode** field of the struct **tmalAdecAc3InstanceConfig_t**. During run-time it can be changed by calling the instance config functions with the command **A3_CONFIG_SET_DUAL_MONO_MODE**. The information on what mode is currently set can be retrieved with the command **A3_CONFIG_GET_DUAL_MONO_MODE**.

tmAdecAc3ConfigTypes_t

```
typedef enum {
   A3_CONFIG_SET_KARAOKE_MODE        = tsaCmdUserBase + 0x00,
   A3_CONFIG_SET_COMP_MODE           = tsaCmdUserBase + 0x01,
   A3_CONFIG_SET_SUBWOOFER_ON        = tsaCmdUserBase + 0x02,
   A3_CONFIG_SET_SUBWOOFER_OFF       = tsaCmdUserBase + 0x03,
   A3_CONFIG_SET_OUTPUT_MODE         = tsaCmdUserBase + 0x04,
   A3_CONFIG_SET_DUAL_MONO_MODE      = tsaCmdUserBase + 0x05,
   A3_CONFIG_SET_DYN_RNG_SCALE_HI    = tsaCmdUserBase + 0x06,
   A3_CONFIG_SET_DYN_RNG_SCALE_LOW   = tsaCmdUserBase + 0x07,
   A3_CONFIG_SET_PCM_SCALE_FACTOR    = tsaCmdUserBase + 0x08,
   A3_CONFIG_GET_KARAOKE_MODE        = tsaCmdUserBase + 0x09,
   A3_CONFIG_GET_COMP_MODE           = tsaCmdUserBase + 0x0a,
   A3_CONFIG_GET_SUBWOOFER_MODE      = tsaCmdUserBase + 0x0b,
   A3_CONFIG_GET_OUTPUT_MODE         = tsaCmdUserBase + 0x0c,
   A3_CONFIG_GET_DUAL_MONO_MODE      = tsaCmdUserBase + 0x0d,
   A3_CONFIG_GET_DYN_RNG_SCALE_HI    = tsaCmdUserBase + 0x0e,
   A3_CONFIG_GET_DYN_RNG_SCALE_LOW   = tsaCmdUserBase + 0x0f,
   A3_CONFIG_GET_PCM_SCALE_FACTOR    = tsaCmdUserBase + 0x10,
   A3_CONFIG_GET_DEF_OUTPUT_MODE     = tsaCmdUserBase + 0x11,
   A3_CONFIG_GET_DEF_SUBWOOFER_MODE  = tsaCmdUserBase + 0x12
} tmAdecAc3ConfigTypes_t;
```

Description

These enum values represent the valid commands for both configuration functions **tmalAdecAc3InstanceConfig** and **tmolAdecAc3InstanceConfig**. All commands of the type **A3_CONFIG_SET_XXX** are used to change an internal setting of the decoder. The commands of the type **A3_CONFIG_GET_XXX** are used to obtain the value of an internal decoder setting. In addition, commands of the type **A3_CONFIG_GET_DEF_XXX** return suggested default settings depending on the packet format of the output descriptor. The parameter field of the **tsaControlArgs_t** struct contains either the value to be set by the decoder (if command is **_SET_**) or it returns the requested value from the decoder (if command is **_GET_**).

For more information refer to **tmalAdecAc3InstanceConfig** on page 86, **tmolAdecAc3InstanceConfig** on page 88, and to *AdecAc3 Configuration* on 46.

tmAdecAc3Capabilities_t

```
typedef struct tmAdecAc3Capabilities{
   ptsaDefaultCapabilities_t   defaultCaps;
   UInt8                       decoderCaps;
} tmAdecAc3Capabilities_t, *ptmAdecAc3Capabilities_t;
```

Fields

`defaultCaps`	Default capabilities. For compliance with the application library architecture, this is a pointer to a structure of the default type.
`decoderCaps`	The 5 least significant bits contain the AC-3 version number to which this decoder is compatible. This number can be compared to the bsid field in the AC-3 bitstream. The decoder is capable of decoding the stream if bsid less or equal decoderCaps.

Description

Structures of this type hold a list of capabilities. The Dolby Digital AC-3 decoder maintains a structure of this type to describe itself. A user can retrieve the address of this structure by calling **tmalAdecAc3GetCapabilties** or **tmolAdecAC3GetCapabilities**.

Note
The AL and OL layers have similar structures; except for the extensions to the default capabilities structure made in the OL layer (tsa.h).

For more information, refer to functions **tmalAdecAc3GetCapabilities** and **tmolAdecAc3GetCapabilities** described in this chapter.

tmalAdecAc3InstanceSetup_t

```
typedef struct tmalAdecAc3InstanceSetup{
   ptsaDefaultInstanceSetup_t     defaultSetup;
   ptmalAdecAc3InstanceConfig_t   ac3Config;
   tmalAdecAc3LibraryMode_t       libraryMode;
   Int                            maxRepeat;
   tmAudioPcmFormat_t             pcmFormatOut0;
   Int                            precisionOut0;
   tmAudioTypeFormat_t            formatOut1;
   Int                            precisionOut1;
   tmAdecAc3StereoOutputMixMode_t stereoMixMode;
   tmAdecAc3DualMonoMode_t        dualMonoMode1;
} tmalAdecAc3InstanceSetup_t, *ptmalAdecAc3InstanceSetup_t;
```

Fields

`defaultSetup`	Refer to tsa.h for more information about this default struct.
`ac3Config`	Pointer to the decoder configuration struct. It contains parameters determining the processing executed by the decoder core.
`libraryMode`	Field to indicate the mode in which the library will be used: **A3_LIB_MODE_PUSH** for frame-based decoding **A3_LIB_MODE_PULL** for streaming mode. These values can be found in **tmalAdecAc3LibraryMode_t.**
`maxRepeat`	Maximum number of blocks that will be repeated before muting.
`pcmFormatOut0`	PCM packet format **dataSubtype** which will be installed by the decoder for the output descriptor of the main multichannel output when the sampling frequency of the AC-3 bitstream is determined.
`precisionOut0`	Number of significant LSBs in the decoded samples. This is used, when the PCM format specified in the above field represents a 32-bit PCM packet type.
`formatOut1`	PCM packet format **dataType** which will be installed by the decoder for the output descriptor of the main multichannel output when the sampling frequency of the AC-3 bitstream is determined.

`precisionOut1`	Number of significant LSBs in the decoded samples. This value is used, only when **formatOut1** is **atfLinearPCM**. The dataSubtype which will be installed depends on the value of this field. It is **apfStereo16** if 16-bit precision is chosen, otherwise it is **apfStereo32**.
`stereoMixMode`	Determines if a normal stereo downmix or a Dolby ProLogic Surround compatible downmix is performed on the second output. The Value of this field matters only when the second output is active.
`dualMonoMode1`	This field determines how dual mono audio data is handled on the second output.

Description

A structure of this type is passed to **tmalAdecAc3InstanceSetup**. The setup function uses this information to do the basic initialization of the AC-3 decoder library (for example, setup of the input and output ports). A pointer to a pre-configured struct of this type can be obtained by calling **tmalAdecAc3GetInstanceSetup**.

The **pcmFormatOut0** and **precisionOut0** fields are used by the decoder during the execution of the instance setup function for internal configuration when no format in the first output descriptor (for outpin 0) is present. Under normal conditions, a format cannot be installed before the decoder is started because the sampling frequency is unknown until it is decoded from the incoming bitstream. Once the sampling frequency is decoded, the output format consisting of the specified PCM type, specified by the **pcmFormatOut0** and **precisionOut0** fields, and the sampling frequency itself, is installed by the decoder by calling the progress function with the flag **tsaProgressFlagChangeFormat**. Supported values for **pcmFormatOut0** are given in Table 6 on page 41.

If an output descriptor for the second output pin is installed, the value of **formatOut1** is used to configure the decoder, when the respective format pointer of the descriptor is NULL. Once the sampling frequency is determined from the AC-3 bitstream, a format is installed for the second output. This format is of the type specified by **formatOut1**. In the case that a PCM stereo format is chosen the **precisionOut1** field determines if the dataSubtype is either **apfStereo16** or **apfStereo32**.

For more information, refer to **tmalAdecAc3LibraryMode_t**, **tmalAdecAc3InstanceSetup**, **tmolAdecAc3InstanceSetup**, and to *TriMedia AC-3 API Overview*, page 14, and *Implementation Aspects*, page 26.

tmalAdecAc3InstanceConfig_t

```
typedef struct tmalAdecAc3Config{
   tmAdecAc3KaraokeMode_t   kCapableMode;
   tmAdecAc3CompMode_t      compMode;
   tmAdecAc3LfeMode_t       outLfeOn;
   tmAdecAc3OutConfig_t     outputMode;
   tmAdecAc3DualMonoMode_t  dualMonoMode;
   Float32                  dynRngScaleHi;
   Float32                  dynRngScaleLow;
   Float32                  pcmScaleFactor;
} tmalAdecAc3Config_t, *ptmalAdecAc3Config_t;
```

Fields

`kCapableMode`	Karaoke-capable reproduction mode. See page 60 for more information.
`compMode`	Dynamic-range compression mode. See page 59 for more information.
`outLfeOn`	Output LFE channel mode. See page 53 for more information.
`outputMode`	Output channel configuration mode. See page 58 for more information.
`dualMonoMode`	Dual mono reproduction mode. See page 61 for more information.
`dynRngScaleHi`	Dynamic range compression cut scale factor (default 1.0). Allowed range: 0.0 ≤ **dynRngScaleHi** ≤ 1.0.
`dynRngScaleLow`	Dynamic range compression boost scale factor (default 1.0). Allowed range: 0.0 ≤ **dynRngScaleLow** ≤ 1.0.

Description

A pointer to a structure of this type is an element of the instance setup structs **tmalAdecAc3InstanceSetup_t** and **tmolAdecAc3InstanceSetup_t**. This information is used to configure the core decoder.

Note
The values of **outputMode** and **outLfeOn** (indicating the presence of the LFE channel) must match the setup for the output of the decoder.

tmolAdecAc3InstanceSetup_t

```
typedef struct tmolAdecAc3InstanceSetup{
   ptsaDefaultInstanceSetup_t     defaultSetup;
   ptmalAdecAc3InstanceConfig_t   ac3Config;
   tmalAdecAc3LibraryMode_t       libraryMode;
   Int                            maxRepeat;
   tmAudioPcmFormat_t             pcmFormatOut0;
   Int                            precisionOut0;
   tmAudioTypeFormat_t            formatOut1;
   Int                            precisionOut1;
   tmAdecAc3StereoOutputMixMode_t stereoMixMode;
   tmAdecAc3DualMonoMode_t        dualMonoMode1;
} tmolAdecAc3InstanceSetup_t, *ptmolAdecAc3InstanceSetup_t;
```

Fields

`defaultSetup`	Refer to tsa.h for more information.
`ac3Config`	Pointer to AC-3 configuration struct.
`libraryMode`	This field is not of importance for the OL layer interface. It exists just for compatibility reasons.
`maxRepeat`	Maximum number of blocks that will be repeated before muting.
`pcmFormatOut0`	PCM packet format **dataSubtype** which will be installed by the decoder for the output descriptor of the main multichannel output when the sampling frequency of the AC-3 bitstream is determined.
`precisionOut0`	Number of significant LSBs if the PCM format specified in the above field represents a 32-bit PCM packet type.
`formatOut1`	PCM packet format **dataType** which will be installed by the decoder for the output descriptor of the main multichannel output when the sampling frequency of the AC-3 bitstream is determined.
`precisionOut1`	Number of significant LSBs if **formatOut1** is **atfLinearPCM**. The dataSubtype which will be installed depends on the value of this field. It is **apfStereo16** if 16-bit precision is chosen, otherwise it is **apfStereo32**.
`stereoMixMode`	Determines if a normal stereo downmix or a Dolby ProLogic Surround compatible downmix is performed on the second output. The Value of this field matters only when the second output is active.

`dualMonoMode1`	This field determines how dual mono audio data is handled on the second output.

Description

A structure of this type is passed to **tmolAdecAc3InstanceSetup**. The decoder is configured based on the values of this struct.

For more information, refer to **tmalAdecAc3InstanceConfig_t** and **tmolAdecAc3InstanceSetup**.

tmAdecAc3HeaderInfo_t

```
typedef struct tmAdecAc3HeaderInfo{
   UInt16                syncWord;
   UInt16                crcWord;
   UInt8                 bsId;
   UInt8                 bsMod;
   tmAdecAc3AcMod_t      acMod;
   UInt8                 cMixLev;
   UInt8                 surMixLev;
   tmAdecAc3SurMod_t     dSurMod;
   tmAdecAc3LfeMod_t     lfeOn;
   UInt8                 dialNorm;
   UInt8                 comprE;
   UInt8                 compr;
   UInt8                 langCodE;
   UInt8                 langCod;
   UInt8                 audProdIE;
   UInt8                 mixLevel;
   tmAdecAc3RoomType_t   roomTyp;
   UInt8                 dialNorm2;
   UInt8                 compr2E;
   UInt8                 compr2;
   UInt8                 langCod2E;
   UInt8                 langCod2;
   UInt8                 audProdi2E;
   UInt8                 mixLevel2;
   tmAdecAc3RoomType_t   roomTyp2;
   tmAdecAc3CopyRight_t  copyrightb;
   tmAdecAc3CopyState_t  origbs;
   UInt8                 timeCod1E;
   UInt16                timeCod1;
   UInt8                 timeCod2E;
   UInt16                timeCod2;
   UInt8                 addbsiE;
   UInt8                 addbsil;
   UInt16                frameSize;
   Float32               sFrequency;
   UInt16                dataRate;
} tmAdecAc3HeaderInfo_t, *ptmAdecAc3HeaderInfo_t;
```

Fields

`syncWord`	Synchronization word.
`crcWord`	First CRC word (start of frame).
`bsId`	Bitstream identification.
`bsMod`	Bitstream mode.
`acMod`	Audio coding mode.

cMixLev	Center mix level.
surMixLev	Surround mix level.
dSurMod	Dolby surround mode.
lfeOn	Low frequency effects channel flag.
dialNorm	Dialog normalization word.
comprE	Compression word exists.
compr	Compression word.
langCodE	Language code exists.
langCod	Language code.
audProdIE	Audio production info exists.
mixLevel	Mixing level
roomTyp	Room type.
dialNorm2	Dialog normalization word #2.
compr2E	Compression word #2 exists.
compr2	Compression word #2.
langCod2E	Language code #2 exists.
langCod2	Language code #2.
audProdi2E	Audio production info #2 exists.
mixLevel2	Mixing level #2.
roomTyp2	Room type #2.
copyrightb	Copyright bit.
origbs	Original bitstream flag.
timeCod1E	Time code first half exists.
timeCod1	Time code first half.
timeCod2E	Time code second half exists.
timeCod2	Time code second half.
addbsiE	Additional BSI exists.
addbsil	Additional BSI length.
frameSize	Size of AC-3 frame in bytes.
sFrequency	Sampling frequency in Hz.
dataRate	Data rate in kbps.

Description

This struct is used to provide the application with information on the properties of the AC-3 bitstream. In streaming mode applications, this information can be obtained by the progress callback function. In non-streaming mode, **tmalAdecAc3DecodeFrame** returns this information in its parameter struct which contains a pointer to a header info struct. Refer also to **tmalAdecAc3Frame_t and tmalAdecAc3DecodeFrame**.

tmalAdecAc3Frame_t

```
typedef struct tmalAdecAc3Frame{
   ptmAvPacket_t            ac3Packet;
   ptmAvPacket_t            pcmPacket;
   Int                      searchRange;
   Int                      offset;
   Int                      frameLength;
   Address                  ancilData;
   tmalAdecAc3HeaderInfo_t  headerInfo;
} tmalAdecAc3Frame_t, *ptmalAdecAc3Frame_t;
```

Fields

`ac3Packet`	Pointer to AC-3 packet.
`pcmPacket`	The decoded PCM samples are written into this audio packet by **tmalAdac3DecodeFrame** and **tmalAdac3MuteFrame**.
`searchRange`	Search range for the function **tmalAdecAc3FindSyncword**. **searchRange** bytes will be searched.
`offset`	Number of bytes from AC-3 data pointer to the first valid sync word. This value will be set by **tmalAdecAc3FindSyncword**.
`frameLength`	Length of the detected frame in bytes.
`ancilData`	Pointer to ancillary data in the AC-3 frame
`headerInfo`	This struct is filled with the information of the current frame by the function **tmalAdecAc3DecodeFrame**.

Description

This structure is used by AL layer applications using the non-streaming (push) mode decoder interface. It is used by the functions **tmalAdecAc3FindSyncword**, **tmalAdecAc3DecodeFrame** and **tmalAdecAc3MuteFrame**.

tmalAdecAc3FindSyncword searches the AC-3 packet data for a valid sync word. The sync word displacement from the data pointer is stored in the **offset** field. This function also writes the length of the frame into the **frameLength** field. The AC-3 packet must contain at least **(searchRange + 4)** bytes of data.

tmalAdecAc3DecodeFrame decodes one AC-3 frame (stored in the AC-3 packet) into one frame of PCM data. The AC-3 packet must contain at least **framelength** bytes of AC-3 data (a complete frame) and the PCM packet buffer must be large enough to receive the PCM packet (1536 * number of channels * sizeOf**Sample** bytes); sizeOfSample is either 2 bytes in 16-bit mode or 4 bytes in 18- or 20-bit mode. This function also updates the **headerInfo** field and the pointer to the ancillary data (**ancilData**).

tmalAdecAc3MuteFrame mutes one frame. The result is written into the PCM packet. The PCM packet buffer must be large enough to receive the PCM packet (1536 × number of channels × sizeOfSample bytes).

For more information, refer to **tmalAdecAc3FindSyncword**, **tmalAdecAc3DecodeFrame**, and **tmalAdecAc3MuteFrame**.

AC-3 API Functions

This section presents the AC-3 Decoder device library functions.

tmalAdecAc3GetCapabilities

```
tmLibappErr_t tmolAdecAc3GetCapabilities(
   ptmAdecAc3Capabilities_t   *caps,
);
```

Parameters

`caps`	Pointer to a variable in which to return a pointer to capabilities data.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function can be used to retrieve a pointer to the capabilities struct of the TriMedia AC-3 decoder library.

For more information, refer to **tmAdecAc3Capabilities_t**.

tmolAdecAc3GetCapabilities

```
tmLibappErr_t tmolAdecAc3GetCapabilities(
   ptmAdecAc3Capabilities_t   *caps,
);
```

Parameters

`caps`	Pointer to a variable in which to return a pointer to capabilities data.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function can be used to retrieve a pointer to the capabilities struct of the TriMedia AC-3 decoder library.

For more information, refer to **tmAdecAc3Capabilities_t**.

tmalAdecAc3Open

```
tmLibappErr_t
tmalAdecAc3Open(
   Int   *instance
);
```

Parameters

`instance`	Pointer (returned) to the instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MALLOC_FAILED`	Memory allocation failure.
`TMLIBAPP_ERR_NO_INSTANCE_AVAILABLE`	No further decoder instance can be instantiated.

Description

Creates an instance of an AC-3 decoder and sets the instance variable. This instance variable must be used in subsequent function calls for this decoder. The open function allocates memory for the internal instance variables.

tmolAdecAc3Open

```
tmLibappErr_t tmolAdecAc3Open(
   Int   *instance
);
```

Parameters

`instance`	Pointer (returned) to the instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MALLOC_FAILED`	Memory allocation failure (only in the OL layer).
`TMLIBAPP_ERR_NO_INSTANCE_AVAILABLE`	No further decoder instance can be opened.

Description

Instantiates an AC-3 decoder and sets the instance variable. This instance variable must be used in subsequent function calls for this decoder. Memory is allocated for internal variables.

tmalAdecAc3Close

```
tmLibappErr_t tmalAdecAc3Clos
   Int   instance,
);
```

Parameters

`instance`	Instance, as returned by **tmalAdecAc3Open**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance.

Description

This function releases the instance of the decoder. It frees the memory allocated for internal variables.

tmolAdecAc3Close

```
tmLibappErr_t tmolAdecAc3Close(
   Int   instance,
);
```

Parameters

`instance`	Instance, as returned by **tmolAdecAc3Open**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance.

Description

Shuts down the instance of the decoder. It frees the memory allocated for internal variables.

tmalAdecAc3GetInstanceSetup

```
tmLibappErr_t tmalAdecAc3GetInstanceSetup(
   Int                            instance,
   ptmalAdecAc3InstanceSetup_t  *setup
);
```

Parameters

`setup`	Pointer to a variable in which to return a pointer to the setup data.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Allocation of memory for the instance setup structure failed.

Description

This function can be used to retrieve a pre-configured AL layer instance setup struct. The structure and all substructures are filled with default settings. The memory allocated by this function is freed by **tmalAdecAc3Close**.

tmolAdecAc3GetInstanceSetup

```
tmLibappErr_t tmolAdecAc3GetInstanceSetup(
   Int                              instance,
   ptmolAdecAc3InstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance, as returned by **tmalAdecAc3Open**.
`setup`	Pointer to a variable in which to return a pointer to the setup data.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function can be used to retrieve a pre-configured OL layer instance setup struct. The structure and all substructures are filled with default settings. When the AC-3 decoder is running, a call to this function will retrieve the settings currently in use.

The debug version of the library triggers an assert if the instance value is incorrect.

tmalAdecAc3InstanceSetup

```
tmLibappErr_t tmalAdecAc3InstanceSetup(
   Int                          instance,
   ptmalAdecAc3InstanceSetup_t  setup
);
```

Parameters

`instance`	Instance, as returned by `tmalAdecAc3Open`.
`setup`	Pointer to the setup data.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number.
`TMLIBAPP_ERR_NO_QUEUE`	Queues for the input pin or for the first output pin are not assigned.
`TMLIBAPP_ERR_UNSUPPORTED_DATACLASS`	Selected data class of input or output format is not supported.
`TMLIBAPP_ERR_UNSUPPORTED_DATATYPE`	Data type of input or output format is not supported.
`TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE`	Data subtype of input or output format is not supported.
`A3_ERR_OUTPUT_MISMATCH`	Conflict between **outputMode** or **outLfeOn** setting in the instance config struct and the format of the output descriptor. The output configuration allows channels that are not supported by the specified packet format.
`A3_ERR_ILL_KARAOKE_MODE`	The **kCapableMode** field of the instance config struct contains an illegal value. See page 60 for the supported values.
`A3_ERR_ILL_COMP_MODE`	The **compMode** field of the instance config struct contains an illegal value. See page 59 for the supported values.
`A3_ERR_ILL_DUAL_MONO_MODE`	The **dualMonoMode** field of the instance config struct contains an illegal value. See page 61 for the supported values.
`A3_ERR_ILL_DYN_RNG_SCALE_HI`	The **dynRngScaleHi** field of the instance config struct contains an illegal value (exceeding the range of [0.0, 1.0]).

A3_ERR_ILL_DYN_RNG_SCALE_LOW	The **dynRngScaleLow** field of the instance config struct contains an illegal value (exceeding the range of [0.0, 1.0]).
A3_ERR_ILL_PCM_SCALE_FACTOR	The **pcmScaleFactor** field of the instance config struct contains an illegal value (exceeding the range of [0.0, 1.0]).

Description

Initializes the decoder using the information in the setup struct. It leaves the decoder in a stopped state. After successful execution of this function the actual decoder can be started by calling **tmalAdecAc3Start** or the respective sequence of non-streaming mode functions.

For more information, refer to **tmalAdecAc3Setup_t**, **tmolAdecAc3Setup_t**, and to *TriMedia AC-3 API Overview* on page 14.

tmolAdecAc3InstanceSetup

```
tmLibappErr_t tmolAdecAc3InstanceSetup(
   Int                  instance,
   ptmolAdecAc3Setup_t   setup
);
```

Parameters

instance	Instance, as returned by **tmolAdecAc3Open**.
setup	Pointer to the setup data.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	Invalid AL layer instance number. OL instance is only checked in debug mode and an assert is triggered in the case that it is invalid.
TMLIBAPP_ERR_NO_QUEUE	Queues for the input pin or for the first output pin are not assigned.
TMLIBAPP_ERR_UNSUPPORTED_DATACLASS	Data class of input or output format is not supported.
TMLIBAPP_ERR_UNSUPPORTED_DATATYPE	Data type of input or output format is not supported.
TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE	Data subtype of input or output format is not supported.
A3_ERR_OUTPUT_MISMATCH	Conflict between **outputMode** or **outLfeOn** setting in the instance config struct and the format of the output descriptor. The output configuration allows channels that are not supported by the specified packet format.
A3_ERR_ILL_KARAOKE_MODE	The **kCapableMode** field of the instance config struct contains an illegal value. See page 60 for the supported values.
A3_ERR_ILL_COMP_MODE	The **compMode** field of the instance config struct contains an illegal value. See page 59 for the supported values.
A3_ERR_ILL_DUAL_MONO_MODE	The **dualMonoMode** field of the instance config struct contains an illegal value. See page 61 for the supported values.

A3_ERR_ILL_DYN_RNG_SCALE_HI	The **dynRngScaleHi** field of the instance config struct contains an illegal value (exceeding the range of [0.0, 1.0]).
A3_ERR_ILL_DYN_RNG_SCALE_LOW	The **dynRngScaleLow** field of the instance config struct contains an illegal value (exceeding the range of [0.0, 1.0]).
A3_ERR_ILL_PCM_SCALE_FACTOR	The **pcmScaleFactor** field of the instance config struct contains an illegal value (exceeding the range of [0.0, 1.0]).

Description

Initializes the decoder using the information in the setup struct. In debug mode asserts are triggered when the OL layer instance is invalid or queues are not assigned. All error messages are returned from the AL layer instance setup function, which is called from the OL layer instance setup function as part of its processing.

After successful setup, the decoder can be started with **tmolAdecAc3Start**.

Refer also to **tmalAdecAc3InstanceSetup** and **tmolAdecAc3InstanceSetup_t**.

tmalAdecAc3InstanceConfig

```
tmLibappErr_t tmalAdecAc3InstanceConfig(
   Int                instance,
   ptsaControlArgs_t  args
);
```

Parameters

`instance`	Instance value returned by **tmalAdecAc3Open**.
`args`	Pointer to the control arguments structure which contains a command and a parameter. The two other fields are not used by this function.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number. The decoder has been either not opened or already been closed.
`TMLIBAPP_ERR_NOT_SETUP`	**tmalAdecAc3InstanceSetup** has not been called yet. The decoder needs to be set up before the config function can be called.
`TMLIBAPP_ERR_INVALID_COMMAND`	The command could not be interpreted by the function.
`A3_ERR_OUTPUT_MISMATCH`	Returned on commands **A3_CONFIG_SET_OUTPUT_MODE** or **A3_CONFIG_SET_SUBWOOFER_ON** if the output configuration would allow channels that are not supported by the packet format specified in the output descriptor.
`A3_ERR_ILL_KARAOKE_MODE`	Returned on command **A3_CONFIG_SET_KARAOKE_MODE** if the mode set in the parameter field is not defined. Refer to 1 for the supported values.
`A3_ERR_ILL_COMP_MODE`	Returned on command **A3_CONFIG_SET_COMP_MODE** if the mode set in the parameter field is not defined. Refer to 1 for the supported values.
`A3_ERR_ILL_DUAL_MONO_MODE`	Returned on command **A3_CONFIG_SET_DUAL_MONO_MODE** if the mode set in the parameter field is not defined. Refer to 1 for the supported values.
`A3_ERR_ILL_DYN_RNG_SCALE_HI`	Returned on command **A3_CONFIG_SET_DYN_RNG_SCALE_HI** if the value

	set in the parameter field exceeds the range of [0.0, 1.0].
A3_ERR_ILL_DYN_RNG_SCALE_LOW	Returned on command **A3_CONFIG_SET_DYN_RNG_SCALE_LOW** if the value set in the parameter field exceeds the range of [0.0, 1.0].
A3_ERR_ILL_PCM_SCALE_FACTOR	Returned on command **A3_CONFIG_SET_PCM_SCALE_FACTOR** if the value set in the parameter field exceeds the range of [0.0, 1.0].
A3_ERR_ILL_CONFIG	The **outputMode** field of the instance config struct contains an illegal value. Refer to 1 for the supported values.

Description

This function is used to either change the configuration of the decoder or obtain information on the current configuration. It is called with a pointer to a **tsaControlArgs_t** struct as argument. This struct contains four elements, two of which are used by the function. The first one is the command, specified in **tmAdecAc3ConfigTypes_t** on page 62. In addition to this, the parameter field is used as input or output dependent on if the command is of the type **_SET_** or **_GET_**. Refer to *AdecAc3 Configuration* on page 46 for more information on the commands and the respective parameters.

tmolAdecAc3InstanceConfig

```
tmLibappErr_t tmalAdecAc3InstanceConfig(
   Int                  instance,
   UInt32               flags,
   ptsaControlArgs_t    args
);
```

Parameters

`instance`	Instance value returned by **tmalAdecAc3Open**.
`flags`	Flags used for the access to the control queue. Typically **tsaControlWait** is used.
`args`	Pointer to the control arguments structure which contains a command and a parameter. The two other fields are not used by this function.

Return Codes

`TMLIBAPP_OK`	Success.
(other return values)	Returned from the function **tsaDefaultInstanceConfig** which is called from this function and implements the queue handling. Those errors indicate problems with the command or response queue or problems with the setup of the component.

Description

This function is used to either change the configuration of the decoder or get information on the current configuration. It is called with a pointer to a **tsaControlArgs_t** struct as argument. This struct contains four elements. The first one is the command, specified in **tmAdecAc3ConfigTypes_t** on 1. The values of certain internal decoder settings can be sent or received in the parameter field. It is used as input or output depending on if the command is of the type **_SET_** or **_GET_**. Refer to *AdecAc3 Configuration* on page 46 for more information on the commands and their parameters. The timeout field is used as timeout value for the access to the response queue. The information if the command has been executed successfully is stored in the retval field. It contains the return value from **tmalAdecAc3InstanceConfig**. Refer to page 86 for the respective error codes.

tmalAdecAc3Start

```
tmLibappErr_t tmalAdecAc3Start(
   Int   instance
);
```

Parameters

`instance`	Instance value returned by **tmalAdecAc3Open**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number. The decoder either has not been opened or has already been closed.
`TMLIBAPP_ERR_NOT_SETUP`	**tmalAdecAc3InstanceSetup** has not been called yet. The decoder needs to be set up before the start function can be called.
`TMLIBAPP_ERR_ALREADY_STARTED`	The decoder is already running. It must be stopped before it can be started again.

Description

Before the decoder can be started, an instance must be opened and the decoder must be initialized using the setup function.

The decoder requires the following callback functions: progressFunction, dataoutFunction, datainFunction, and errorFunction. Refer to *AdecAc3 Progress* on page 45 for more information on the behavior of the progress function and to *AdecAc3 Errors* on page 44 for the error function.

The decoder continues to run until the application calls the stop function. The decoder checks to see if it has been requested to stop at several points during the decoding process. If the decoder determines that it has been asked to stop, it returns all the packets that it has in its possession, and then returns.

For more information, refer to **tmalAdecAc3Stop**, and *TriMedia AC-3 API Overview* on page 14. Also refer to Book 3, *Software Architecture*.

tmolAdecAc3Start

```
tmLibappErr_t tmolAdecAc3Start(
   Int   instance
);
```

Parameters

`instance`	Instance value returned by **tmolAdecAc3Open**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_TCREATE_FAILED`	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.

Description

Before you can start the decoder, you must open an instance and initialize the decoder using the setup function.

The decoder makes use of the progress and error callback functions to interact with the application or with other components. Providing implementations for those functions is optional. However, in an application using real-time audio playback, we recommend that you determine the format of the AC-3 bitstream using the progress callback function and then set up the audio hardware accordingly.

It is also useful to provide an error function to detect problems with the input bitstream. The decoder is capable of handling all error situations after data streaming is started. Before the start of data streaming, the decoder checks that:

- The instance is correct.
- The setup function has been called.
- The current instance of the decoder is already started.

In all those cases, the error callback function is called with the respective error message (refer to *AdecAc3 Errors* on page 44). After returning from the error callback function, the decoder task is terminated. It is important for the application to know of this, so that it can resolve the problem. Otherwise, it would assume that the decoder is still running in its own context.

When your application calls **tmolAdecAc3Stop**, the decoder returns all the packets that it has in its possession.

It is possible to set up the instance again, after the decoder has been stopped. After that, the same instance of the decoder can be restarted. It can also be restarted without calling

tmolAdecAc3InstanceSetup after its stopping. When the decoder is running in its own context, internal parameters can be changed or examined by using **tmolAdecAc3InstanceConfig**.

In addition to the previously described error messages the **tmolAdecAc3Start** function triggers asserts in debug mode when the OL layer instance is incorrect or the decoder is not set up.

For more information, refer to **tmalAdecAc3InstanceSetup**, **tmolAdecAc3InstanceSetup**, **tmalAdecAc3Stop**, **tmolAdecAc3Stop**, and *TriMedia AC-3 API Overview* on page 14. Also refer to Book 3, *Software Architecture*.

tmalAdecAc3Stop

```
tmLibappErr_t tmalAdecAc3Stop(
   Int   instance,
);
```

Parameters

`instance`	Instance returned by **tmalAdecAc3Open**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number. The decoder either has not been opened or has already been closed.
`TMLIBAPP_ERR_NOT_SETUP`	The decoder has not been set up yet.

Description

Stops the decoder by setting a stop flag. The function immediately returns with **TMLIBAPP_OK**. This function is used only when the decoder is operated in streaming mode at the AL layer. It is otherwise not directly called from the application.

Note that the decoder can continue to run for additional cycles after the call to this stop function, because it recognizes the stop flag only after loading the first part of the frame, after loading the second part of the frame, and after finishing the whole frame.

Before the decoder stops, it returns all AC-3 and PCM packets that it still has in its possession.

For more information, refer to **tmalAdecAc3Start**, and *TriMedia AC-3 API Overview* on page 14. Also refer to Book 3, *Software Architecture*.

tmolAdecAc3Stop

```
tmLibappErr_t tmolAdecAc3Stop(
   Int   instance,
);
```

Parameters

`instance`	Instance returned by **tmalAdecAc3Open** or **tmolAdecAc3Open**.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Stops the decoder by executing the default stop sequence. The decoder is notified of the stop request by receiving a stop message on either the input pin or one of the output pins. It expels all data packets that it keeps. When **tmolAdecAc3Stop** returns, the decoder task is not destroyed, but suspended. It can be resumed by calling **tmolAdecAc3Start**. The task gets destroyed by **tmolAdecAc3Close**.

In debug mode this function can assert **TMLIBAPP_ERR_INVALID_INSTANCE** and **TMLIBAPP_ERR_NOT_SETUP**.

For more information, refer to **tmalAdecAc3Start**, **tmolAdecAc3Start**, and *TriMedia AC-3 API Overview* on page 14. Also refer to Book 3, *Software Architecture*.

tmalAdecAc3FindSyncword

```
tmLibappErr_t tmalAdecAc3FindSyncword(
   Int                  instance,
   ptmalAdecAc3Frame_t   ac3Frame
);
```

Parameters

instance	Instance value returned by **tmalAdecAc3Open**.
ac3Frame	Pointer to a struct that contains all other parameters required to search for a syncword.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	Invalid instance number. The decoder either has not been opened or has already been closed.
A3_ERR_INVALID_PARAMETER	**ac3Frame** is a null pointer, the pointer to the AC-3 data (in the AC-3 packet) is a null pointer, or the search range value is zero or negative.
A3_ERR_NOT_ENOUGH_DATA	The AC-3 packet does not contain enough data to search the complete searchrange. There must be at least **searchrange** bytes in the AC-3 packet.
A3_ERR_SYNC_NOT_FOUND	There was no valid sync word in the AC-3 packet within the search range.
A3_ERR_ILL_DATA_RATE	A sync word bit pattern was found, but it is probably false because the frame size code of the frame is illegal.
A3_ERR_ILL_SAMPLE_RATE	A sync word bit pattern was found, but it is probably false because the sample rate code of the frame is illegal.

Description

This function searches a packet of AC-3 data for a valid sync word.

The argument **ac3Frame** points to a struct that is used by all non-streaming (push) mode processing functions. **tmalAdecAc3FindSyncword** uses the fields **ac3Packet** and **searchRange** of this struct as its input and writes its results (if it found a valid sync word) into the fields **offset** and **frameLength**. It also stores the sample rate of the detected AC-3 frame into the sFrequency field of the **tmAdecAc3HeaderInfo_t** struct to which a pointer is stored in **ac3Frame**.

The AC-3 packet must contain at least **searchrange** bytes of AC-3 data. The search range must be larger than zero. Otherwise, **tmalAdecAc3FindSyncword** returns an error.

If **tmalAdecAc3FindSyncword** finds a valid sync word, it writes the sync word offset (number of bytes) from the AC-3 data pointer into the **ac3Frame–>offset** field.

If **tmalAdecAc3FindSyncword** does not find a valid sync word, it returns an error.

See also: **tmalAdecAc3Open**, **tmalAdecAc3Close**, **tmalAdecAc3InstanceSetup**, **tmalAdecAc3InstanceConfig**, **tmalAdecAc3MuteFrame**, **tmalAdecAc3Frame_t**, and *TriMedia AC-3 API Overview* on page 14, and *Implementation Aspects* on page 26.

tmalAdecAc3DecodeFrame

```
tmLibappErr_t tmalAdecAc3DecodeFrame(
   Int                  instance,
   ptmalAdecAc3Frame_t   ac3Frame
);
```

Parameters

instance	Instance value from **tmalAdecAc3Open**.
ac3Frame	Pointer to a struct that contains all parameters to decode one AC-3 frame into one PCM frame.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	Invalid instance number.
TMLIBAPP_ERR_NOT_SETUP	**tmalAdecAc3InstanceSetup** has not been called.
A3_ERR_NOT_ENOUGH_DATA	The **ac3Packet** in the **ac3Frame** struct does not contain a complete frame.
A3_ERR_PCM_BUFFER_TO_SMALL	The buffer in the PCM packet is not large enough to receive one complete frame.
A3_ERR_PUSH_DEC_FATAL	Fatal error while decoding. No output data will be produced. The application must take appropriate action (e.g. muting).
A3_ERR_PUSH_DEC_NON_FATAL	The decoder had problems decoding the current frame. It repeated or muted one or more blocks. A complete frame with PCM samples is stored in **ac3Frame–>pcmPacket**. This is just a warning.
A3_ERR_SYNC_NOT_FOUND	There was no valid sync word in the AC-3 packet at the memory position calculated from the AC-3 data pointer plus **offset**.
A3_ERR_ILL_DATA_RATE	The frame size code of the AC-3 frame is illegal.
A3_ERR_ILL_SAMPLE_RATE	The sample rate code of the frame is illegal.
A3_ERR_CRC1_FAILED	The first cyclic redundancy check failed. The entire PCM frame is invalid in this case.
A3_ERR_CRC2_FAILED	The second cyclic redundancy check failed. In this case the first two PCM blocks (512 samples) are valid. The remaining 4 blocks contain invalid data.

Description

This function decodes one frame of AC-3 data into one frame of PCM data.

To decode one AC-3 frame, there must be one complete frame in the AC-3 packet in **ac3Frame**. To find a starting point of a complete frame in the bit steam, the function **tmalAdecAc3FindSyncword** can be used. It detects the start of an AC-3 frame and calculates the length of this frame. The application must put this frame into the AC-3 packet (the pointer **ac3Frame–>ac3Packet–>buffers[0].data + offset** must point to the beginning of the frame).

If the AC-3 packet contains an entire frame, the decoder can process the frame and put the resulting PCM data into the PCM packet. A pointer to a PCM packet is an element of the argument structure **ac3Frame**. The buffer in the PCM packet must be large enough to receive one complete PCM frame. The required size (in bytes) can be calculated:

#decoded channels × A3_SAMPLES_PER_CHAN_FRAME × SIZE_OF_SAMPLE

The number of decoded channels is determined by the output configuration (**tmalAdecAc3InstanceSetup**). It can be 1, 2, 4 or 6 channels. **A3_SAMPLES_PER_CHAN_FRAME** is a constant that is defined in *tmalAdecAc3.h* and represents the number of samples per channel for one decoded AC-3 frame (1,536). The output of the decoder is either 16-bit or 32-bit linear PCM. Therefore, the number of samples must be multiplied by **SIZE_OF_SAMPLE** which is 2 or 4 depending on the packet format's data subtype. If the buffer is not large enough to receive a complete frame of PCM data, the decoder returns an error.

If a fatal error occurs while decoding, no output data will be generated. In this case, an error will be returned. The application must then perform an appropriate error handling, which could be muting or repeating of the previous frame.

For more information, refer to **tmalAdecAc3Open**, **tmalAdecAc3Close**, **tmalAdecAc3InstanceSetup**, **tmalAdecAc3InstanceConfig**, **tmalAdecAc3FindSyncword**, **tmalAdecAc3MuteFrame**, **tmalAdecAc3Frame_t**, as well as the sections *TriMedia AC-3 API Overview* on page 14, *The AL Layer* on page 16, and *Implementation Aspects* on page 26.

tmalAdecAc3MuteFrame

```
tmLibappErr_t tmalAdecAc3MuteFrame(
   Int                  instance,
   tmalAdecAc3Frame_t   ac3Frame
);
```

Parameters

`instance`	Instance value returned by **tmalAdecAc3Open**.
`ac3Frame`	Pointer to a structure that contains all other parameters to mute one frame.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number.

Description

This function mutes one frame (1,536 samples per channel) of audio data. It can be used when the decoder returned a fatal error, or if the application must produce silence.

The buffer in the PCM packet must be large enough to receive one complete PCM frame. The required size (in bytes) can be calculated:

decoded channels × A3_SAMPLES_PER_CHAN_FRAME × SIZE_OF_SAMPLE

The number of decoded channels is determined by the output configuration (as performed by **tmalAdecAc3InstanceSetup**. It can be 1, 2, 4, or 6 channels. **A3_SAMPLES_PER_CHAN_FRAME** is a constant that is defined in tmalAdecAc3.h and represents the number of samples per channel for one decoded AC-3 frame (1,536). The output of the decoder is either 16-bit or 32-bit linear PCM. Therefore, the number of samples must be multiplied by 2 or 4. If the buffer is not large enough to receive a complete frame of PCM data **tmalAdecAc3MuteFrame** returns an error.

For more information, refer to **tmalAdecAc3Open**, **tmalAdecAc3Close**, **tmalAdecAc3InstanceSetup**, **tmalAdecAc3InstanceConfig**, **tmalAdecAc3FindSyncword**, **tmalAdecAc3MuteFrame**, **tmalAdecAc3Frame_t**, as well as the sections *TriMedia AC-3 API Overview* on page 14, *The AL Layer* on page 16, and *Implementation Aspects* on page 26.

Chapter 13

Pro Logic Decoder (AdecPl) 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. In addition, this algorithm is owned by Dolby Labs and an appropriate license must be obtained for its use. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

Introduction

Dolby Pro Logic Surround is an audio coding technique that transmits and stores stereo-compatible multichannel audio. This is achieved by a mix of four channels into two at the encoder side. As a result, the Pro Logic encoded audio can be listened to whether a Pro Logic decoder is present or not. This is not the case for digitally compressed audio like MPEG or AC-3 (Dolby Digital). In such cases a decoder is required for listening to the audio.

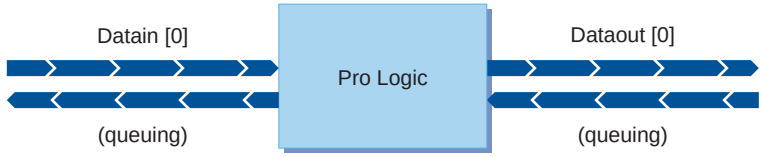


Figure 16 Structure of the Pro Logic decoder

Dolby Pro Logic Surround is widely used for VHS cassettes, laser disks, PC games and TV applications. It allows for movie theatre 3D sound positioning in a home environment. Dolby Pro Logic Surround is derived from the older Dolby Surround technique. The main difference is the better separation of the audio channels accomplished by using an adaptive decoding matrix as opposed to a passive one as used by Dolby Surround.

The original implementation of Dolby Surround and Dolby Pro Logic was done with analog circuitry. Nowadays, this task can easily be performed by DSPs and general purpose processors. The load of a TriMedia for this task is less than 9 MIPS.

Dolby Pro Logic Surround will continue to play an important role in the future even though the digital compression algorithms like AC-3 and MPEG multichannel provide better channel separation and require less transmission bandwidth. AC-3 for instance explicitly supports a special Pro Logic two channel coding mode and it is a requirement for certain AC-3 decoder products to be capable to decode Pro Logic as well.

Principles of the Pro Logic Encoder

The Dolby Pro Logic Surround encoder mixes a four channel input signal (left, center, right and surround) to a two channel output signal (left-total and right-total). Figure 17, following, depicts the processing implemented by a Dolby Surround encoder.

The center channel is simply attenuated by 3 dB and added to both the left and the right channel. A bit more effort is required for the surround channel before it is mixed with the left and right channel (already containing the center).

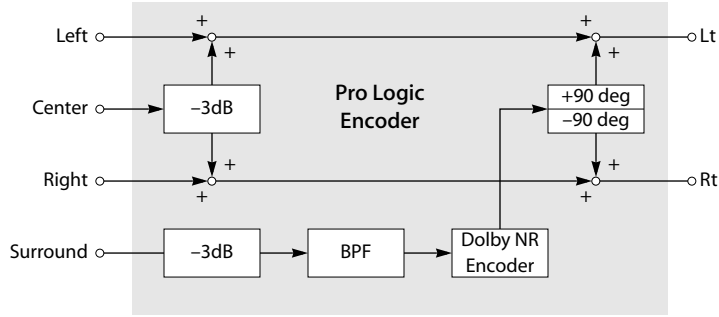


Figure 17 Block Diagram of Dolby Surround Encoder

The surround channel is also attenuated by 3 dB before it is bandpass filtered (100 Hz to 7 kHz). After that, the surround signal is encoded by a Dolby noise reduction filter block. The resulting signal is then phase-shifted by 90 degrees. Finally, the output of the phase shifter is added to the left channel and subtracted from the right channel. The resulting two signals form the Pro Logic encoded audio signal.

The special processing on the surround channel is done to achieve better channel separation in the decoder. If Lt and Rt were perfect transmission channels, a decoder would need only add Lt and Rt to retrieve the center channel and subtract Rt from Lt to retrieve the surround channel. No crosstalk between center and surround would occur. However, real-world transmission channels cause signal leakage from the center into the surround. The effect can be reduced by limiting the bandwidth of the surround channel and applying the noise reduction encoding to it.

Principles of the Pro Logic Decoder

A Dolby Pro Logic Surround decoder must reproduce a four-channel surround image as well as possible. It must not only invert the processing of the encoder, but also minimize the effects of unwanted crosstalk between the channels. For this reason, the Pro Logic decoder uses an adaptive matrix dynamically updated during decoding based on the properties of Lt, Rt, (Lt + Rt) and (Lt – Rt).

The main signal path of the decoder is illustrated in Figure 18.

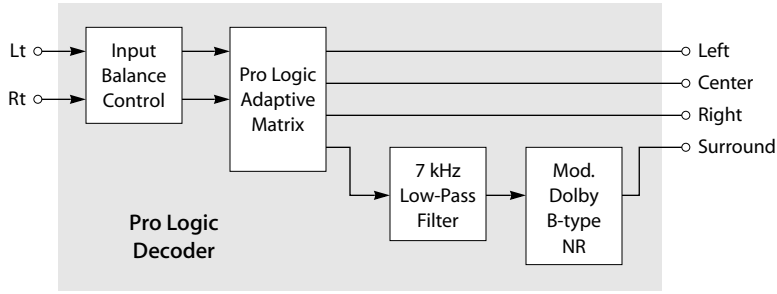


Figure 18 Block Diagram of the Channel Decoding Path of the Pro Logic Decoder

The first step of the decoder is scaling the two input signals. If Lt and Rt are in balance, the decoder attenuates both signals by 3 dB. If there is a mismatch, the decoder adapts the incorrect level applying autobalance processing. The three output channels Left, Center, and Right are then reproduced by linear combinations of Lt' and Rt' using weight factors stored in the adaptive matrix. As in the encoder, the surround channel requires some extra signal processing in the form of a low pass filter and a Dolby B-type noise reduction filter.

If the elements of the adaptive matrix remain unmodified, the processing is the same as the processing of a Dolby Surround decoder. Updating the matrix elements is based on the detection of signal dominance. It is possible for the control block to determine that a dominant signal exists in the room spanned by the four cardinal directions left, center, right and surround. If the decoder detects a dominant signal, it suppresses leakage into other channels by adapting the matrix elements. The dominance detection block is also capable of switching off directional enhancements when no dominant signal is found. In that case, the decoder behaves like a passive Dolby Surround Decoder.

For an example of a dominant signal, consider an audio stream that must create the effect of a helicopter flying in a circle above the listener. The dominant direction is the direction of the helicopter. The intention is to emphasize this one position in the room and to provide proper localization. If multiple dominant sounds exist, the Pro Logic decoder would fail to provide such localization. Therefore, the mixing process on the encoder side must be carried out carefully.

An example where no directional enhancement is desirable is the sound of rain or wind, intended to come from all directions. Special localization is generally not desirable.

Special Considerations of the TriMedia Implementation

The block diagram illustrated in Figure 18 does not show all processing blocks that are required for a full functional surround audio decoding. For instance, a delay line for the surround channel is required to prevent sound, intended to come from a front speaker,

from arriving at the listener earlier from the surround channel due to leakage effects. The exact delay time depends on the positions of the loud speakers in the room and, of course, on the position of the listener. The time delay and some other features required for a complete Dolby Pro Logic Surround solution are not part of the TriMedia Pro Logic Decoder library. They are implemented in other TriMedia audio system components like the Audio Mixer library. An example how a complete audio system could look like is given on page 111.

The TriMedia Pro Logic decoder library provides three options to influence the decoder processing shown in Figure 18.

- The autobalance control can optionally be switched off. If switched off, both input signals are equally attenuated by 3 dB.
- The signal processing applied to the surround output can optionally be bypassed. This mode is called wide surround mode and it is applicable in multimedia PC applications.
- If the output format supports two surround channels, the surround channel produced by the core decoder is split into two channels. Before the actual split, it is attenuated by 3 dB to maintain constant loudness.

Overview of the TriMedia Pro Logic Decoder Library

The TriMedia Pro Logic Library comes in form of a TSA library. It can be used at both the AL and OL layer. The provided functionality is the same for both API levels. Depending on the application's requirements it has to be decided what API is appropriate. Philips recommends the use of the OL layer interface because its ease of use. In some special cases the AL layer interface may be preferable. It has the disadvantage that mechanisms for data exchange between different software components must be implemented in the context of the application, as opposed to being hidden in the library in case of the OL layer interface.

The Pro Logic decoder consists of one input pin accepting data packets with several PCM audio formats and one output pin sending decoded multichannel audio packets in different PCM formats. In the setup phase of the decoder a certain format has to specified for the input and output pin.

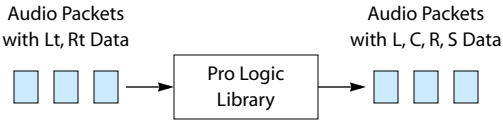


Figure 19 Inputs and Outputs of the Pro Logic Decoder Library

Supported Packet Formats

Following PCM audio packet subtypes are supported by the TriMedia Pro Logic Decoder library at its input:

Table 17 Supported Packet Formats

16 bit formats	32 bit formats
apfStereo16	apfStereo32
apfFourCh_3_1_0_16	apfFourCh_3_1_0_32
apfFourCh_2_2_0_16	apfFourCh_2_2_0_32
apfFourCh_2_1_1_16	apfFourCh_2_1_1_32
apfFourCh_3_0_1_16	apfFourCh_3_0_1_32
apfFiveDotOne16	apfFiveDotOne32

Lt and Rt are always supposed to be at the position of the left and right channel within the input data packets. The decoder just uses those samples and neglects the remaining channels.

At its output the library supports the same packet types as at is input except for the stereo formats. There is one constraint on the output packet format. It must provide all channels decoded by the decoder. The output channel configuration is determined during the setup phase by the config structure field **chanconfig** (see page 119). The output consists of left and right channel plus a combination of center and one or two surround channels.

Decoder Configurations

During the setup phase of the decoder the application can determine what parts of the decoder are active. The instance setup struct contains a pointer to the configuration struct which consists of four elements.

- **abaldisable**
  If 0, the input balance block of Figure 18 is active. If 1, both input signals are attenuated by 3 dB.
- **chanconfig**
  Determines how many channels are produced by the decoder. Left and Right are always present at the output. The presence of center and surround(s) depends on the value of this field (See page 119).
- **widesur**
  If 0, the low pass filter and the modified B-type noise reduction filter are applied to the surround channel. If 1, the surround output of the adaptive matrix stays unfiltered.
- **pcmScaleFactor**
  Floating point number that is used to scale the samples of all output channels.

All the above described decoder settings can also be changed dynamically during the processing phase by using **tmolAdecPlInstanceConfig** if the OL interface is used or **tmalAdecPlInstanceConfig** if the AL interface is used, respectively.

Note that the sample rate of the input descriptor's format determines the characteristics of the filters implemented in the decoder. If the sample rate is set to 0.0 the default value 48.0 kHz is used.

Using the OL Layer API

The usage of the Pro Logic decoder library at the OL layer is similar to that of other OL layer libraries. At first an instance of the decoder is obtained by calling **tmolAdecPlOpen** and the decoder capabilities are retrieved from the decoder by calling **tmolAdecPlGetCapabilities**. The decoder capabilities are required by **tsaDefaultInOutDescriptorCreate** to properly set up the input and output descriptors. Then a prototype of the instance setup struct required to configure the decoder instance is obtained by calling **tmolAdecPlGetInstanceSetup**. The application does the necessary modifications of the instance setup struct and uses it then to configure the Pro Logic decoder by calling **tmolAdecPlInstance-**

Setup. This setup provides the decoder with pointers to callback functions and the information on the configuration of the input and output pins. After that the static configuration of the library is done. The actual decoding can be started using **tmolAdecPlStart**. From now on, the decoder operates as a separate task in its own context. Communication with the application and other OL component is implemented by callback functions. In addition to this, the application or a different library can change the decoder configuration or acquire information on it by sending commands to the decoder task. These commands are sent by calling **tmolAdecPlInstanceConfig**. The Pro Logic decoder can be stopped by calling **tmolAdecPlStop**. This function forces the decoder to expel all internally held packets and leave the main processing loop. Finally, the decoder instance can be released by calling **tmolAdecPlClose**.

Constraints on Input/Output Packets

Aside from the restriction on the packet format types (see page 104) there are some constraints on the packet sizes. The Pro Logic decoder works internally on chunks of 8 samples. To simplify the internal buffer management, all full input packets must contain a multiple of 8 samples per channel. On the output side, empty packets are accepted only if they can accommodate at least eight samples across all channels. There is no direct dependency between the input and output packet sizes. They are independent of each other at the OL layer. Note that this is different for pure AL layer applications!

Following table contains the number of bytes that must be present in a full input packet or allocated in an empty output packet. All multiples of those values are valid as well.

Table 18 Granularity of Input/Output Packets

packet format	full input packet data size (in bytes)	empty output packet buffer size (in bytes)
apfStereo16	32	not supported
apfFourCh_X_X_X_16	64	64
apfFiveDotOne16	96	96
apfStereo32	64	not supported
apfFourCh_X_X_X_32	128	128
apfFiveDotOne32	192	192

Note that the size of the input and output packet influences the real time behavior of the decoder:

- Decoding delay: the size of the output packet determines the delay of the decoder. In applications where a low latency is required the output packet size must be chosen accordingly.
- MIPS consumption: the decoder works more efficiently with larger packets because they reduce the amount of internal control overhead. If the processor load is a critical matter, both input and output packets should be chosen large enough.

- To optimize the decoder, the smallest possible packet format should be chosen. That is always a stereo format for the input. At the output, it depends on whether surround splitting is desired.

Time Stamps

The TriMedia Pro Logic Decoder is capable of handling time stamps. It assigns the newest valid time stamp received with the input packets to the current output packet at the moment the output packet is filled and sent back via the dataout callback function. If the granularity of the input packets is coarser than the granularity of the output packets, only the first output packet of a group being related to the current input packet would obtain the input packet's time stamp. The time stamps of the successive output packets would be marked invalid. The decoder does not adapt its processing to the time stamps, nor does it interpret them. They are presentation time stamps and the application has to ensure that the decoder gets its input decoder at the appropriate time.

Run Time Behavior

Once the function tmolAdecPlStart is called the Pro Logic decoder starts to run as a separate task. All the interaction with other components and the application occurs via callback functions. The decoder makes use of the datain, dataout and error function. Whenever a fatal error occurs the respective error code is sent via the error callback function. All empty output and full input packets are expelled and the AL layer start function is left.

The decoder works without any internal buffers. It writes the decoded samples directly into the output packets.

Changes to the configuration can be done by calling **tmolAdecPlInstanceConfig**. This function sends a message to the decoder via the command queue. Since the commands are checked only when data input or output occurs, configuration changes do not happen at the same time when the config function is called. The delay of a configuration change depends on the sizes of the input and output packets, because those determine the frequency at which the datain and dataout callback functions are called by **tmalAdecPlStart**.

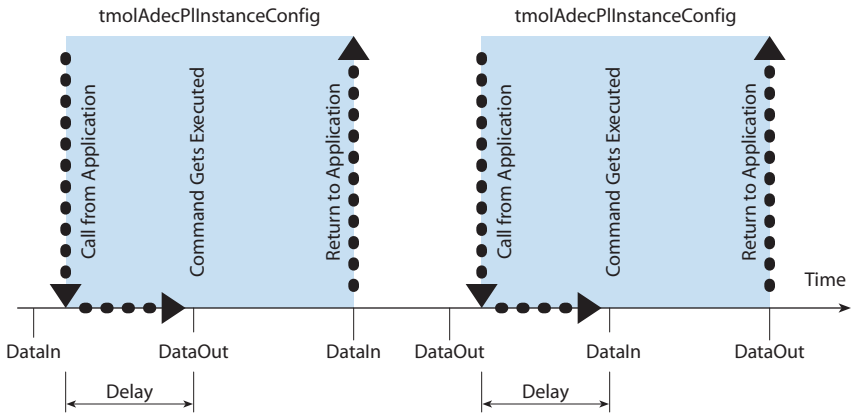


Figure 20 Time Behavior of tmolAdecPlInstanceConfig

Using the AL Layer API

At the AL layer a programmer has two choices of interfaces. It is possible to write an AL layer application in streaming (pull) mode and non-streaming (push) mode. Philips recommends the usage of the non-streaming mode interface at the AL layer. However, the TriMedia Pro Logic Decoder is also fully functional in streaming mode at the AL layer.

Operation in Streaming Mode

If the AL layer interface is used in streaming mode, application programmers must implement their own callback functions. The default mechanisms are only available for OL layer applications. The code required to use the streaming mode AL layer library is very similar to that of the OL layer. At first an instance is obtained by calling **tmalAdecPlOpen**. Then, the instance of the decoder is configured with an instance setup struct. A pointer to an already configured struct can be retrieved from **tmalAdecPlGetInstanceSetup**. It is then adapted to the application's requirements and used to configure the decoder by **tmalAdecPlInstanceSetup**. After that the actual decoding is started by calling **tmalAdecPlStart**. From now on the decoder acquires and sends data automatically using the respective callback functions. It can be stopped by calling **tmalAdecPlStop**. If no operating system is used, the call of the stop function must occur either in an interrupt service routine or in one of the callback functions. Once the decoder is stopped the current instance of it can be freed by **tmalAdecPlClose**. An example of how to implement an application using the AL layer streaming mode API is provided with the software release. The name of the example file is *exaladecpls.c*.

Operation in Non-Streaming Mode

To use the TriMedia Pro Logic Decoder in non-streaming mode the steps required for acquiring an instance and setting up the instance are the same as for the streaming mode application. The actual processing is done by the function **tmalAdecPlDecode**. This function decodes one chunk of Lt, Rt input data. It has as parameter a struct of the type **tmalAdecPlFrame_t**. This struct consists of two pointers, one to an input packet and the other to an output packet. A block diagram of the actual processing loop is given in Figure 21. Shaded boxes represent functionality to be implemented in the application and white boxes are functions provided by the Pro Logic library.

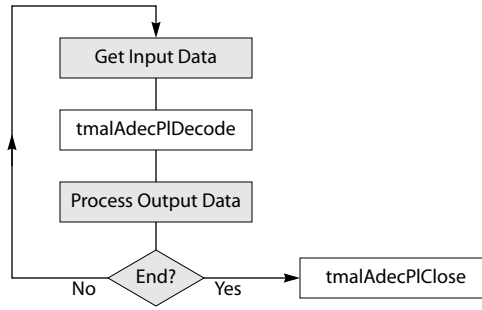


Figure 21 Processing Loop in Non-Streaming Mode

An example program using the non-streaming mode AL layer API is given by *exaladec-plns.c* which is shipped together with the library.

Constraints on Input/Output Packets

If the application is using the streaming mode API of the AL layer, the same constraints as for the OL layer apply (see page 106). In non-streaming mode there apply two restrictions. First of which is that the number of PCM samples of each channel must be a multiple of eight. In addition to this the output packet buffer size must be large enough to store all decoded samples of the input packet.

The performance considerations made for the OL layer library hold for the AL layer library, too: larger buffers decrease the required processor load for the decoding; however, they also increase the decoding delay.

Time Stamps

The handling of time stamps is the same for the streaming mode interface as for the OL layer library (see page 107). In non-streaming mode time stamps are not handled. A user of the non-streaming mode API can just copy the time stamp information from the input to the output packet, since their associated time duration is the same.

Run Time Behavior

If the application is using the AL layer API in streaming mode it must provide at least a datain and a dataout function. An error function is not necessarily required, since when an error occurs **tmalAdecPlStart** returns with the respective error code and the application can react in an appropriate way. In contrast to the OL layer config function its AL layer counterpart **tmalAdecPlInstanceConfig** changes the configuration immediately upon its call. No delay occurs.

For the non-streaming mode API no particular run time issues exist. The run time behavior is completely determined by the application, as it is in charge of input/output synchronization and buffer management.

Quality Assurance and Performance

This section describes how the quality and functionality of the Pro Logic decoder library is assured. Furthermore, the library performance is also analyzed.

Quality Assurance

The quality of the Pro Logic library is assured by design of the software and an extensive test suite. The decoder design was based upon the reference C code revision number 2.0 available from Dolby Labs. During the entire implementation and optimization phase, extensive compliance tests were performed. A set of 140 test vectors testing all critical decoder properties at different sampling frequency was used to assure compliance. Dolby provides a set of Matlab tools that generate objective measures telling whether or not an individual test failed. These tests were carried out with the optimized TriMedia implementation. Since then all tests were carried out with an automatic test environment comparing all decoding results to those proved as correct with the Matlab tests.

The TriMedia ProLogic decoder has been completely tested and certified by Dolby Labs.

Apart from the testing of the implemented algorithm comprehensive tests of the TSSA interface are performed to guarantee that the library is capable of dealing with all specified packet types.

Furthermore, extensive listening tests were carried out with different applications built upon the Pro Logic library.

Decoder Performance

The processor resources requirements have been measured for all four supported sample rates with an AL and an OL layer Pro Logic decoder application. In both cases the decoder's input packet format is **apfStereo16** and the output packet format **apfFiveDotOne16**.

The performance measure unit is MIPS which is independent of the TriMedia clock frequency. A 100 MHz TriMedia executes 100 million VLIW instructions and therefore provides 100 MIPS.

Table 19 Performance Measurement Results

Sample Rate	AL Layer	OL Layer
22050 Hz	3.7 MIPS	4.0 MIPS
32000 Hz	5.4 MIPS	5.8 MIPS
44100 Hz	7.4 MIPS	8.0 MIPS
48000 Hz	8.1 MIPS	8.7 MIPS

The values of Table 19 are worst-case figures for the specified sampling frequencies.

Note that the AL layer results do not take the handling of input and output data into account. The OL layer results, however, represent the entire time spent in the Pro Logic decoder task. This overhead in the OL layer result increases with the sampling frequency, because the amount of data travelling through the decoder increases.

Additional Requirements For a Complete Audio System

The functionality of the Pro Logic decoder is limited to the actual decoding algorithm. Additional features, that are absolutely required for a real audio application, must be implemented external to the Pro Logic decoder. An example for a real audio system build upon available TriMedia TSA libraries is illustrated by Figure 22.

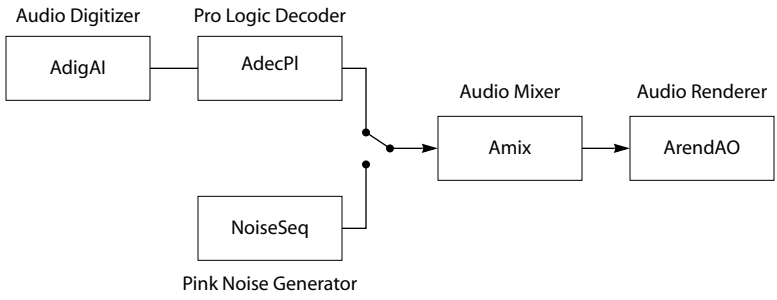


Figure 22 Pro Logic Decoder Audio System

The tasks of the connected components are:

- Audio Digitizer: Samples PCM data from the audio input hardware and sends it as TSA audio packets to the Pro Logic Decoder.
- Pink Noise Generator: Produces pink noise which is used to calibrate the audio system dependent on the location of the speakers.

- Pro Logic Decoder: Decodes the left and right channel samples received in the input packets. It sends the surround sound PCM samples at its output to the audio mixer.
- Audio Mixer: Receives the multichannel audio packets at its input. It applies additional processing like master volume control, trim, bass redirection and center/surround delays.
- Audio Renderer: Receives audio packets from the mixer and renders the sound via the audio output hardware. It is also capable of handling time stamps to ensure that the audio packets are displayed at the right time.

AdecPl Inputs and Outputs

The Pro Logic Decoder library has got one input and one output. Several different PCM audio formats are supported by the Pro Logic Decoder library. Details can be found on page 104.

The restrictions on the inputs and outputs of the Pro Logic Decoder library are described for the OL layer on page 106, and for the AL layer on page 109, respectively.

AdecPl Errors

Following errors can be reported by the error callback function during the actual data processing of the library (only in streaming mode!). All those errors are fatal and the AL layer start function terminates after the error callback function has returned to it. Before the AL layer start function returns, it expels all packets it kept.

The error callback function, which must be implemented by the application programmer, should inform the application about the fact that the data processing will stop after it returns. The application can then, for instance, restart the Pro Logic decoder.

See page 107 and 110 for further information on the run time behavior of the Pro Logic Library.

Error Codes	Meaning
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance number is invalid.
`TMLIBAPP_ERR_NOT_SETUP`	Instance is not yet setup.
`TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE`	Format of input or output packet is not supported.
`PL_ERR_IN_GRANULARITY`	Number of samples per channel in the input packet is not a multiple of eight.

AdecPl Progress

The TriMedia Pro Logic Decoder library does not make use of the progress callback function. The exception is, of course, the installation of a new format at its output. See the respective chapter about the TriMedia Streaming Software Architecture for more details.

AdecPl Configuration

The TriMedia Pro Logic Decoder library provides two configuration functions, one for OL layer applications and one for AL layer applications. The OL layer configuration function uses command queues. Its run time behavior is described on page 107, refer also to page 142 for its interface description. On the contrary the AL layer configuration function works without command queues. The commands are executed directly. Refer to page 110 for information on its run time behavior and to page 130 for its interface description.

Both function accept the same commands which either change or get the values of the auto balance mode, the wide surround mode, the channel configuration mode, and the scale factor of the output samples.

Commands to change the configuration are:

`PL_CONFIG_SET_AUTOBALANCE_ON`	Switches the auto balance input control on. Parameter is not used.
`PL_CONFIG_SET_AUTOBALANCE_OFF`	Switches the auto balance input control off. Parameter is not used.
`PL_CONFIG_SET_CHANNEL_CONFIG`	Changes the output channel configuration. Parameter is in the range [3, 7]. For the meaning of the values, refer to n.
`PL_CONFIG_SET_SAMPLERATE`	Changes the sample rate, which affects the internal filters. Parameter is in the range [0, 3] where 0 = 48000 Hz, 1 = 44100 Hz, 2 = 32000 Hz, and 3 = 22050 Hz.
`PL_CONFIG_SET_WIDE_SURROUND_ON`	Switches wide surround mode on (see page 105). Parameter is not used
`PL_CONFIG_SET_WIDE_SURROUND_OFF`	Switches wide surround mode off (see page 105). Parameter is not used
`PL_CONFIG_SET_PCM_SCALE_FACTOR`	Sets the factor with which the output samples are multiplied. Parameter is in the range [0.0, 1.0]

Commands to get values of the current configuration are:

`PL_CONFIG_GET_AUTOBALANCE_MODE`	Parameter returns the current status of the auto-balance mode, parameter == 0, auto balance is switched off parameter == 1, auto balance is switched on.
`PL_CONFIG_GET_CHANNEL_CONFIG_MODE`	Parameter returns the current channel configura-

	tion mode. Refer to n for the meaning of the codes.
PL_CONFIG_GET_SAMPLERATE	Parameter returns the current sample rate, where 0 = 48000 Hz, 1 = 44100 Hz, 2 = 32000 Hz, and 3 = 22050 Hz.
PL_CONFIG_GET_WIDE_SURROUND_MODE	Parameter returns the current surround mode. Parameter == 0 means normal surround channel processing; parameter == 1 means LPF and B-type NR filter switched off.
PL_CONFIG_GET_PCM_SCALE_FACTOR	Parameter returns the current PCM scale factor, a floating point value of the range [0.0, 1.0]

The (one) command to get default values is:

PL_CONFIG_GET_DEF_CHANNEL_CONFIG	Parameter return the default channel configuration depending on the format of the output descriptor (refer to).

If the application wants either to change the value of the PCM scale factor or retrieve the current setting from the running decoder a special cast operation is required. The following code fragments show how those commands must be set up. In the first example the PCM scale factor is set to 0.8:

```
tsaControlArgs_t cargs;
Float32          pcmVal = 0.8;
cargs.command   = PL_CONFIG_SET_PCM_SCALE_FACTOR;
cargs.parameter = *((Pointer *) &pcmVal);
tmolAdecAc3InstanceConfig( decInstance, tsaControlWait, &cargs);
```

If the application wants to acquire the current setting of this decoder parameter it has to perform the following:

```
tsaControlArgs_t cargs;
Float32          pcmVal;
cargs.command   = PL_CONFIG_GET_PCM_SCALE_FACTOR;
tmolAdecAc3InstanceConfig( decInstance, tsaControlWait, &cargs);
pcmVal = *((Float32 *) &cargs.parameter);
```

This sort of casting is required because otherwise an implicit cast to integer would be performed by the compiler.

Pro Logic AL Layer API Data Structures

This section describes the Pro Logic data structures.

tmalAdecPlLibraryMode_t

```
typedef enum {
   PL_LIB_MODE_PUSH,
   PL_LIB_MODE_PULL
} tmalAdecPlLibraryMode_t;
```

Description

The decoder can be run in push mode or in pull mode (only in the AL layer). These are legal values for the field **libraryMode** in **tmalAdecAc3InstanceSetup_t**. Push mode is also known as non-streaming mode. Pull mode is also known as streaming mode.

For more information, refer to structures **tmalAdecPlSetup_t** (see page 118) and **tmalAdecPlInstanceSetup**(see page 125).

tmalAdecPlConfigTypes_t

```
typedef enum {
   PL_CONFIG_SET_AUTOBALANCE_ON       = tsaCmdUserBase + 0x00,
   PL_CONFIG_SET_AUTOBALANCE_OFF      = tsaCmdUserBase + 0x01,
   PL_CONFIG_SET_CHANNEL_CONFIG       = tsaCmdUserBase + 0x02,
   PL_CONFIG_SET_SAMPLERATE           = tsaCmdUserBase + 0x03,
   PL_CONFIG_SET_WIDE_SURROUND_ON     = tsaCmdUserBase + 0x04,
   PL_CONFIG_SET_WIDE_SURROUND_OFF    = tsaCmdUserBase + 0x05,
   PL_CONFIG_SET_PCM_SCALE_FACTOR     = tsaCmdUserBase + 0x06,
   PL_CONFIG_GET_AUTOBALANCE_MODE     = tsaCmdUserBase + 0x07,
   PL_CONFIG_GET_CHANNEL_CONFIG_MODE  = tsaCmdUserBase + 0x08,
   PL_CONFIG_GET_SAMPLERATE           = tsaCmdUserBase + 0x09,
   PL_CONFIG_GET_WIDE_SURROUND_MODE   = tsaCmdUserBase + 0x0a,
   PL_CONFIG_GET_PCM_SCALE_FACTOR     = tsaCmdUserBase + 0x0b,
   PL_CONFIG_GET_DEF_CHANNEL_CONFIG   = tsaCmdUserBase + 0x0c
} tmalAdecPlConfigTypes_t;
```

Description

These are the valid commands for both the AL and OL layer instance config function. They can be used to either change an internal decoder setting **PL_CONFIG_SET**_*xxx*, get the current setting **PL_CONFIG_GET**_*xxx* or get a default value **PL_CONFIG_GET_DEF**_*xxx*.

For more information, refer to the functions **tmalAdecPlInstanceConfig** (see page 130), and **tmolAdecPlInstanceConfig** (see page 142).

tmalADecPlCapabilities_t

```
typedef struct tmalADecPlCapabilities{
   ptsaDefaultCapabilities_t   defaultCaps;
} tmalADecPlCapabilities_t, *ptmalAdecPlCapabilities_t;
```

Fields

`defaultCaps`	Pointer to TSA default capabilities structure.

Description

Structures of this type hold a list of capabilities. This audio decoder maintains a structure of this type to describe itself. A user can retrieve the address of this structure by calling **tmalAdecPlGetCapabilities**.

tmalAdecPlSetup_t

```
typedef struct tmalAdecPlSetup{
   ptsaDefaultInstanceSetup_t   defaultSetup;
   ptmalAdecPlConfig_t          config;
   tmalAdecPlLibraryMode_t      libraryMode;
} tmalAdecPlSetup_t, *ptmalAdecPlSetup_t;
```

Fields

`defaultSetup`	Pointer to **tsaDefaultInstanceSetup_t**.
`config`	Pointer to **tmalAdecP1Config_t**.
`libraryMode`	enum value determining whether AL library is operated in streaming or non-streaming mode.

Description

This structure consists of a pointer to a **tsaDefaultInstanceSetup_t** struct, a pointer to a config struct (**tmalAdecP1Config_t**) and an **integer** field which indicates whether the library is used in streaming or non-streaming mode. Structs of this type are used to configure the decoder. While the default setup describes the interface of the decoder to the outside world (callback functions and input/output pins) the internal behavior of the decoder is determined by the config structure.

A valid instance setup structure with pre-configured settings can be obtained from the Pro Logic decoder library by calling **tmalAdecPlGetInstanceSetup** (see page 125). The content of the setup struct is used to configure the decoder by calling **tmalAdecPlInstanceSetup** (see page 126) with a pointer to the struct as second parameter.

tmalAdecPlConfig_t

```
typedef struct tmalAdecPlConfig{
   Int     abaldisable;
   Int     chanconfig;
   Int     widesur;
   Float   pcmScaleFactor;
} tmalAdecPlConfig_t, *ptmalAdecPlConfig_t;
```

Fields

`abaldisable`	Disable autobalance.
`chanconfig`	Output channel configuration.
`widesur`	Wide surround mode.
`pcmScaleFactor`	PCM output sample scale factor.

Description

This struct is used to configure the decoder during the setup phase. The Pro Logic instance setup struct see page 118) contains a pointer to the config structure. Its content determines the algorithmic behavior of the decoder:

- **abaldisable**: If 1, auto balance is disabled; the default setting is enabled (therefore 0).
- **chanconfig**: Determines the output configuration of the decoder, as indicated in the table below.

Table 20 Channel Configuration Codes

chanconfig code	reproduced audio channels
3 = 3/0	left, center, right
4 = 2/1	left, right and one surround + phantom center
5 = 3/1	left, center, right and one surround
6 = 2/2	left, right and split surrounds + phantom center
7 = 3/2	left, center, right and split surrounds

The default setting depends on format of the output descriptor.

Table 21 Default Channel Configuration Values

Output Descriptor Format	Default Setting
apfFourCh_3_0_1_16	3
apfFourCh_2_1_1_16	4

Table 21 Default Channel Configuration Values

Output Descriptor Format	Default Setting
apfFourCh_3_0_1_16	3
apfFourCh_3_1_0_16	5
apfFourCh_2_2_0_16	6
apfFiveDotOne16	7

- **widesur**: If set, the low pass filter and the Dolby B-type NR filter are not performed on the samples of the surround channel. This is recommended for PC type applications. Default wise these filters are applied.
- **pcmScaleFactor**: A value of the range [0.0, 1.0] with which all PCM output samples are multiplied. The default value is 1.0, i.e., the output samples stay unmodified.

tmalAdecPlFrame_t

```
typedef struct tmalAdecPlFrame{
   ptmAvPacket_t   PlpcmPacket;
   ptmAvPacket_t   pcmPacket;
} tmalAdecPlFrame_t, *ptmalAdecPlFrame_t;
```

Fields

`PlpcmPacket`	Pointer to input packet.
`xpcmPacket`	Pointer to output packet.

Description

Structures of this type consist of two pointers. Both are pointing to data packets; the first to an input packet and the second to an output packet. This structure is used as argument for the function **tmalAdecPlDecode**, which implements the non-streaming mode decoder.

Pro Logic AL layer API Functions

This section describes the Pro Logic Application Layer API functions.

tmalAdecPlGetCapabilities

```
extern tmLibappErr_t tmalAdecPlGetCapabilities(
   ptmalAdecPlCapabilities_t   *caps
);
```

Parameters

`caps`	Pointer to a variable in which to return a pointer to the capabilities data.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Function returns pointer to the decoder capabilities struct via the argument pointer.

tmalAdecPlOpen

```
extern tmLibappErr_t tmalAdecPlOpen(
   Int   *instance
);
```

Parameters

`instance`	Pointer (returned) to the instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NO_INSTANCE_AVAILABLE`	No further instance is free.

Description

Returns instance if a free instance is available via the argument pointer.

tmalAdecPlClose

```
extern tmLibappErr_t tmalAdecPlClose(
   Int   instance
);
```

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is invalid.

Description

Closes the instance and frees resources.

tmalAdecPlGetInstanceSetup

```
extern tmLibappErr_t tmalAdecPlGetInstanceSetup(
   Int                  instance,
   ptmalAdecPlSetup_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.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is invalid.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation failed.

Description

The function provides a prototype of a valid instance setup struct. It fills in the most likely values. Memory is allocated dynamically.

Note
The queues of the input and output descriptors need to be assigned in both push and pull model.

tmalAdecPlInstanceSetup

```
extern tmLibappErr_t tmalAdecPlInstanceSetup(
   Int                  instance,
   ptmalAdecPlSetup_t   setup
);
```

Parameters

instance	Instance variable.
setup	Pointer to setup data.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	Instance is invalid.
TMLIBAPP_ERR_NO_QUEUE	The queues for the input or output pins are not assigned.
TMLIBAPP_ERR_UNSUPPORTED_DATACLASS	The dataClass field in either the format of the input or output descriptor contains an unsupported value.
TMLIBAPP_ERR_UNSUPPORTED_DATATYPE	The dataType field in either the format of the input or output descriptor contains an unsupported value.
TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE	The dataSubype field in either the format of the input or output descriptor contains an unsupported value.
PL_ERR_ILL_SAMPRATE	The sample rate specified in the format of the input descriptor is not supported.
PL_ERR_ILL_CHCNFG	The value of the **chanconfig** field exceeds the range [3, 7].
PL_ERR_OUTPUT_MISMATCH	The channel configuration setting and the format of the output descriptor contain conflicting values, i.e., the channel configuration setting would produce audio channels that are not present in the specified output format.
PL_ERR_ILL_PCMSCALEFACTOR	The PCM sample scale factor exceeds the interval [0.0, 1.0].

Description

Function copies information from the Pro Logic setup structure into an internal structure belonging to the instance and checks if the settings are valid.

Note
The sample rate value of the input format is used to determine the internal filter characteristics.

tmalAdecPlStart

```
extern tmLibdevErr_t tmalAdecP1Start(
   Int   instance
);
```

Parameters

instance	The instance.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	Instance is invalid.
TMLIBAPP_ERR_NOT_SETUP	Instance is not yet setup.
TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE	The format of input or output packet is not supported.
PL_ERR_IN_GRANULARITY	The number of samples per channel in the input packet is not a multiple of eight.

Description

Function works in a loop mode, it acquires empty PCM output data packets and full Pro Logic PCM input packets. The condition for the packets is, that the data size of the input packets and the buffer size of the output packets is always a multiple of 8 samples. The number of samples in the input and output packets does not need to be identical. The decoder writes its results directly into the output packets. Therefore, no memory will be allocated during decoding. The loop processing can be stopped by calling the AL layer stop function **tmalAdecPlStop**.

tmalAdecPlStop

```
extern tmLibappErr_t tmalAdecPlStop(
   Int   instance
);
```

Parameters

`instance`	Instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is invalid.

Description

Sets internal flag that is checked by the start function. The start function then terminates at the next possible position.

tmalAdecPlInstanceConfig

```
extern tmLibappErr_t tmalAdecPlInstanceConfig(
   Int                instance,
   ptsaControlArgs_t  args
);
```

Parameters

`instance`	Instance variable obtained from open function.
`args`	Pointer to argument structure consisting of a command and a parameter pointer. The two other fields are unused at the AL layer.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is invalid.
`TMLIBAPP_ERR_NOT_SETUP`	Instance is not yet setup.
`PL_ERR_ILL_CHCNFG`	**chanconfig** value is illegal.
`PL_ERR_ILL_SAMPRATE`	Sample rate code is not supported.
`PL_ERR_OUTPUT_MISMATCH`	**chanconfig** and format of output descriptor conflict with another.
`PL_ERR_ILL_PCMSCALEFACTOR`	The PCM sample scale factor exceeds the interval [0.0, 1.0].
`TMLIBAPP_ERR_INVALID_COMMAND`	Command is unknown.

Description

This function is used to change or to get information on the operational mode of the decoder. This function needs not be called if the default settings meet the application's requirements. It can be used during the data processing phase to change the decoder configuration without stopping the decoder. It is applicable in both streaming and non-streaming mode. In both cases the configuration is changed before the function returns. There is no special mechanism implemented to synchronize changes with the start function.

This function can execute the commands specified in **tmalAdecPlConfigTypes_t** (refer to 116). One of those commands are assigned to the command field of the args struct. Depending of the nature of the command it could be necessary to assign a value to the parameter field. This is the case for **PL_CONFIG_SET_CHANNEL_CONFIG**, **PL_CONFIG_SET_SAMPLERATE** and **PL_CONFIG_SET_PCM_SCALE_FACTOR**.

Supported parameter values are:

- `PL_CONFIG_SET_CHANNEL_CONFIG`

parameter	reproduced audio channels
3 = 3/0	left, center, right
4 = 2/1	left, right and one surround + phantom center
5 = 3/1	left, center, right and one surround
6 = 2/2	left, right and split surrounds + phantom center
7 = 3/2	left, center, right and split surrounds

- `PL_CONFIG_SET_SAMPLERATE`

parameter
0 = 48000 Hz
1 = 44100 Hz
2 = 32000 Hz
3 = 22050 Hz

- **PL_CONFIG_SET_PCM_SCALE_FACTOR**
  The parameter must be of the range [0.0, 1.0].

If the command is a 'get' command, the requested value is returned in the args struct in the parameter field.

Note
Parameter is of the type **Pointer** and a cast to **integer** or **float** must be performed in the application program because the Pro Logic Decoder Library does not use the parameter value as a pointer, it interprets the content of the field as **integer** or **float** value.

tmalAdecPlDecode

```
extern tmLibappErr_t tmalAdecPlDecode(
   Int                  instance,
   ptmalAdecPlFrame_t   PlFrame
);
```

Parameters

`instance`	Instance variable.
`PlFrame`	Pointer to a PlFrame struct.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is invalid.
`TMLIBAPP_ERR_NOT_SETUP`	Instance is not yet setup.
`TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE`	The format of input or output packet is not supported.
`PL_ERR_IN_GRANULARITY`	The number of samples per channel in the input packet is not a multiple of eight.
`PL_ERR_INOUT_MISMATCH`	The buffer size of the output packet is not great enough to accommodate all decoded samples.

Description

Decoder function for non-streaming mode. This function has no internal buffer. The number of samples of the input and output packets must match (refer to error codes above). If, for instance, the dataSize of the input packet is 256, the input dataSubtype is **apfStereo16**, and the output packet dataSubtype is **apfFiveDotOne16**, then the bufferSize of the output packet must greater equal 768.

Note
The Pro Logic Decoder Library uses only one buffer in both input and output packets.

Pro Logic Operating System Layer API Data Structures

This section presents the Pro Logic Operating System Layer API data structures.

tmolAdecPlCapabilities_t

```
typedef struct tmolADecPlCapabilities{
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmolADecPlCapabilities_t, *ptmolAdecPlCapabilities_t;
```

Fields

`defaultCapabilities`	Pointer to the default capabilities struct as specified in *tsa.h*.

Description

Structures of this type hold a list of capabilities.This audio decoder maintains a structure of this type to describe itself. A user can retrieve the address of this structure by calling **tmolAdecPlGetCapabilities**.

tmolAdecPlInstanceSetup_t

```
typedef struct tmolAdecPlInstanceSetup_t {
   ptsaDefaultInstanceSetup_t   defaultSetup;
   ptmalAdecPlConfig_t          config;
} tmolAdecPlInstanceSetup_t, *ptmolAdecPlInstanceSetup_t;
```

Fields

`defaultSetup`	Pointer to the default instance setup struct. This struct contains the information that is required for the decoder to interact with the application and other library components.
`config`	Pointer to the Pro Logic Decoder configuration struct. It contains settings that determine the sort of signal processing applied to the input data.

Description

This structure consists of two pointers. The first points to a **tsaDefaultInstanceSetup_t** struct. The second points to the AL layer user configuration struct **tmalAdecPlConfig_t** which is used to control the channel mapping of the decoder output and the implemented signal processing (wide surround, auto balance). A pre-configured struct of the type **tmolAdecPlInstanceSetup_t** can be retrieved from the decoder library by calling **tmolAdecPlGetInstanceSetup** (see page 141). The user must then adapt all necessary settings to the application's requirements. The decoder gets finally configured by calling **tmolAdecPlInstanceSetup** with a pointer to a **tmolAdecPlInstanceSetup_t** struct as second parameter (see page 139).

Pro Logic Operating System Layer API Functions

This section presents the Pro Logic Operating System Layer API functions.

Note
All functions trigger asserts in debug mode. They do not check the validity of the instance and if the instance is set up in release mode! It is therefore recommended to use the debug version of the library during the development phase.

tmolAdecPlGetCapabilities

```
extern tmLibappErr_t tmolAdecPlGetCapabilities(
   ptmolAdecPlCapabilities_t   *pCap
);
```

Parameters

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

Return Codes

TMLIBAPP_OK	Success.

Description

Fills in the pointer to a static **tmolADecCapabilities_t** structure maintained by the decoder to describe the capabilities and requirements of this library.

tmolAdecPlOpen

```
extern tmLibappErr_t tmolAdecPlOpen(
   Int   *instance
);
```

Parameters

`instance`	Pointer (returned) to the instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Allocation of memory for the internally used instance variable failed.

Description

Creates an instance of a decoder. Memory for an instance structure and a setup structure is allocated.

tmolAdecPlClose

```
extern tmLibappErr_t tmolAdecPlClose(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolAdecPlOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INST`	The instance is invalid.

Description

Shut down this instance of the decoder. Free instance variable memory.

tmolAdecPlInstanceSetup

```
extern tmLibappErr_t tmolAdecPlInstanceSetup(
   Int                          instance,
   ptmolAdecPlInstanceSetup_t   setup
);
```

Parameters

instance	The instance.
setup	Pointer to setup data.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	The instance is invalid.
TMLIBAPP_ERR_NO_QUEUE	The queues for the input or output pins are not assigned.
TMLIBAPP_ERR_UNSUPPORTED_DATACLASS	The dataClass field in either the format of the input or output descriptor contains an unsupported value.
TMLIBAPP_ERR_UNSUPPORTED_DATATYPE	The dataType field in either the format of the input or output descriptor contains an unsupported value.
TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE	The dataSubype field in either the format of the input or output descriptor contains an unsupported value.
PL_ERR_ILL_SAMPRATE	The sample rate specified in the format of the input descriptor is not supported. Note that the samprate field of the config struct has lower importance! The sample rate is determined by the format of the input descriptor!
PL_ERR_ILL_CHCNFG	The value of the chanconfig field exceeds the range [3, 7].
PL_ERR_OUTPUT_MISMATCH	The channel configuration setting and the format of the output descriptor contain conflicting values, i.e. the channel configuration setting would produce audio channels that are not present in the specified output format.
PL_ERR_ILL_PCMSCALEFACTOR	The PCM sample scale factor exceeds the interval [0.0, 1.0].

Description

This function configures the Pro Logic Decoder. All information required for the set up and configuration of the Pro Logic decoder is contained in the setup struct (see page 135). Normally the setup struct is obtained from the library by calling **tmolAdecPlGetInstanceSetup**. It is then modified by the application to match the application's requirements before **tmolAdecPlGetInstanceSetup** gets called.

tmolAdecPlGetInstanceSetup

```
tmLibdevErr_t tmolAdecPlInstanceSetup(
   Int                          instance,
   ptmolAdecPlInstanceSetup_t  *rsetup
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is invalid.

Description

This function is used to obtain a valid instance setup structure from the library. The library fills in the most likely settings. In the best case, an application program just needs to fill in pointers to the input, output and the control descriptors in the respective descriptor fields before calling **tmolAdecPlInstanceSetup**. This function does not allocate memory for the instance setup struct. The allocation happens when an instance is acquired by calling **tmolAdecPlOpen**.

tmolAdecPlInstanceConfig

```
tmLibdevErr_t tmolAdecPlInstanceConfig(
   Int                  instance,
   UInt32               flags,
   ptsaControlArgs_t    args
);
```

Parameters

`instance`	Instance value from **tmolAdecPlOpen**.
`flags`	Flags for the pSOS command queue.
`args`	Pointer to argument structure containing a command (see **tmalAdecPlConfigTypes_t**), a parameter pointer, a return value pointer (retval) and a timeout value. Note that the AL layer instance config function's return value is stored in the return value pointer parameter!

Return Codes

`TMLIBAPP_OK`	Success.
other	Error codes from the command queue handler implemented in the default functions.

Description

This function is used to either reconfigure particular settings of the decoder or get their values from the decoder. The functionality is identical to that of **tmalAdecPlInstanceConfig** (see page 130). In fact, **tmolAdecPlInstanceConfig** makes use of its AL layer counterpart. The command and the parameter are sent via a command queue to the AL layer function which then executes the command. In the case of a **PL_CONFIG_GET**_*xxx* command the requested value is stored in the parameter field of the args struct. Similarly to the AL layer function, type casts need to be used to access the values.

The OL layer InstanceConfig function works in a synchronized fashion. It can exchange information with its AL layer counterpart, where the actual processing occurs, only at certain moments. Those are when data input or data output occurs. Hence, the delay time of the execution of a command depends on the data granularity.

The value of the timeout field of the args struct determines the number of pSOS clock ticks the function should wait for the response from **tmalAdecPlInstanceConfig** on the response queue. This value is only of importance if the flag **tsaControlWait** is set.

Note
The application using **tmolAdecPlInstanceConfig** must check both the return code of the function and the **retval** field of the args struct. The return value of **tmolAdecPlInstanceConfig** just indicates whether or not the communication via the command queue was successful. Error codes of the command execution are stored in the **retval** field.

tmolAdecPlStart

```
extern tmLibdevErr_t tmolAdecPlStart(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolAdecPlOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
other	Return values from the default start function.

Description

Starts the AL layer start function as task, which implements the actual Pro Logic decoding. The data processing can be stopped by calling **tmolAdecPlStop**. All errors occurring during the execution of the AL layer start function are made known to the application by the error callback function. The return values of **tmalAdecPlStart** (see page 128) are sent as the **errorCode** via the error callback function. After sending the error message **tmalAdecPlStart** returns.

tmolAdecPlStop

```
extern tmLibdevErr_t tmolAdecPlStop(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolAdecPlOpen**.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function triggers the stop sequence for the instance of the Pro Logic decoder. It forces the decoder to leave the main processing loop and return from **tmolAdecPlStart**.

Chapter 14

MPEG Audio Decoder (AdecMpeg) 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.

Overview

Introduction

The MPEG audio decoder is a TSSA compliant module that accepts a stream of MPEG 1 layer 1 and layer 2 encoded audio at its input stream and generates a linear PCM format output stream. It is also able to handle the respective MPEG-2 bit streams. However, it decodes only the stereo channels of MPEG-2 streams. For information about the general interface philosophy, you are directed to the TSSA software architecture documentation.

The public programmers interface of the decoder is the file tmolAdecMpeg.h. This TriMedia library does not support a non-streaming interface. Therefore, no AL header file is made public.

Use of either of these decoders may require a patent license, as the MPEG audio coding standards are covered by patents held by various companies.

MPEG Compliancy

The decoder is capable of decoding all Layer 1and Layer 2 bit streams except for bit streams using the free data rate format. Such bit streams cause an error message. The decoder is also not performing de-emphasis. It, however, indicates if emphasis is applied to MPEG bit stream via the progress callback function when the appropriate flag is installed.

Inputs and Outputs

The decoder has one input and two outputs. The input is an MPEG 1 encoded bit stream. The first output is stereo 16 bit linear PCM audio data, as described by a TSA packet. Stereo 16 bit is the only supported output format. The sample rate can be 32k, 44.1k, or 48k, as described by the MPEG specification. The second will support IEC601937 formatted data, or a headphone mix, in the future.

Real Time Behavior

This section describes some issues of using the decoder in a real time application as buffering, time stamping, and synchronization.

Input/Output Buffering

The MPEG-1 audio decoder accepts TSSA data packets of the type atfMpeg and sends out packets of the type atfLinearPCM and the subtype apfStereo16. On its input side the decoder implements a flexible buffer management. It accepts packets of any size. On the output side, however, it accepts only packets that can accommodate at least one frame of

decoded audio which is 284 samples for Layer 1 and 1152 samples for Layer 2. The decoder sends the output packet when it is filled with one decoded audio frame. It does not try to fill the rest of the packet with data from successive frames.

Time Stamps

The MPEG audio decoder is capable of attaching time stamps to the PCM data packets which are copied from the incoming MPEG packets. It is ensured that the time stamps are assigned to the correct PCM packets.

Synchronization

After the start function of the decoder has been called the decoder can either be in sync or out of sync. It reports a change of this state through the progress function if the progress flag ADEC_MPEG1_PROG_REPORT_FIND_SYNC is installed. Whenever the decoder is not in sync it is not producing audio output. It loses the synchronization, when settings in the MPEG headers change, the header is invalid, the distance to the next frame is incorrect, or the optional CRC is incorrect. In the latter three cases an error is reported via the error callback function. In all cases the progress function is called if the above mentioned progress flag is installed.

The decoder does not perform any muting or block repeating when it loses sync. It is up to downstream components to implement features like that.

Errors

The errors reported by the MPEG decoder are all defined in tmolAdecMpeg.h. The base value of these errors is 0x140A0000, as defined in tmLibappErr.h.

The user can install a TSA standard error callback function, and the decoder will call this if it encounters errors while decoding the bit stream. In that case, the errorCode will be one of the values defined in tmolAdecMpeg.h. Errors reported by the error function are not fatal, and processing will continue as the decoder attempts to recover from the error.

Apart from the standard TSSA errors that are defined in tmLibappErr.h the following component specific errors can occur during the execution of the start function:

`ADEC_MPEG1_ERR_INVALID_HEADER` The ID bit in the MPEG header equals zero.

`ADEC_MPEG1_ERR_FREE_FORMAT_NOT_SUPPORTED` MPEG bit stream does not have a specified data rate. This mode is not supported.

`ADEC_MPEG1_ERR_LAYER3_NOT_SUPPORTED` Decoder can only handle Layer 1 and 2 bit streams.

`ADEC_MPEG1_ERR_CRC_FAILED` The calculation of the cyclic redundancy check failed. This is an indication for a corrupted bit stream and/or transmission errors.

ADEC_MPEG1_ILLEGAL_FRAME_LENGTH	The decoder read more bits than permitted by the standard to decode the last frame. This is an indication that either the encoder did not work properly or that transmission errors occurred.

Progress

The user can install a TSA standard progress callback function. The decoder will use this in several cases.

1. To report a change in format, per standard TSSA behavior. The defaults handle this.
2. To report a change in format to the user. In this case, the progress flag is **ADEC_MPEG1_PROG_REPORT_FORMAT**, and the progress argument description field is a pointer to a data structure of the type **tmAdecMpegFormat_t**.
3. To report the state of the decoder while decoding. In this case, the progress flag is **ADEC_MPEG1_PROG_REPORT_FIND_SYNC**. The decoder reports its state in the description field of the progress arguments struct. It contains a pointer to an integer. The integer value is either **DECODER_NOT_IN_SYNC** or **DECODER_IN_SYNC**. Note that the progress function only reports transitions between these two states.
4. To report that a frame is decoded successfully. In this case, the progress flag is **ADEC_MPEG1_PROG_REPORT_EVERY_FRAME**. This can be used to count frames or to do some performance measurements.

Configuration

Although the decoder does export the standard configuration function, no configuration changes are supported.

Using the MPEG Audio Decoder API

The TriMedia MPEG Audio decoder API is contained within the archived application library libtmAdecMpeg.a. For OL layer applications, you must include the tmolAdecMpeg.h header file. AL layer operation is not supported.

The OL Layer

The operating system layer only supports data streaming operation. A diagram of the typical flow of control is shown in Figure 23.

The capabilities of the component should be obtained using **tmolAdecMpegGetCapabilities**. This information will be used by the format manager to ensure that the two instances being connected together are compatible. An instance of the audio decoder should be obtained using **tmolAdecMpegOpen**. InOutDescriptors which connect the

audio decoder to other components should be created by initializing **ptsaInOutDescriptorSetup_t** structures and calling **tsaDefaultInOutDescriptorCreate** for each connection. This function can also be used to automatically create packets which will be used to transfer data between component instances.

The pointer to the audio decoder instance setup should be obtained using **tmolAdecMpegGetInstanceSetup**. This structure should be initialized with any application specific values. The application should then call **tmolAdecMpegInstanceSetup** to configure the instance.

Data streaming can then be initiated by calling **tmolAdecMpegStart**. Coded audio packets to be decoded are obtained using the datain call back function which is provided in the tsaDefaults library. An output packet will be obtained using the dataout call back function and this will be used to store the decoded audio data.

The application can terminate data streaming using **tmolAdecMpegStop**, and release the instance using **tmolAdecMpegClose**. After the instance has been closed, the application should destroy the InOutDescriptor using the **tsaDefaultInOutDescriptorDestroy** function. This will automatically free the packets contained in the queues.

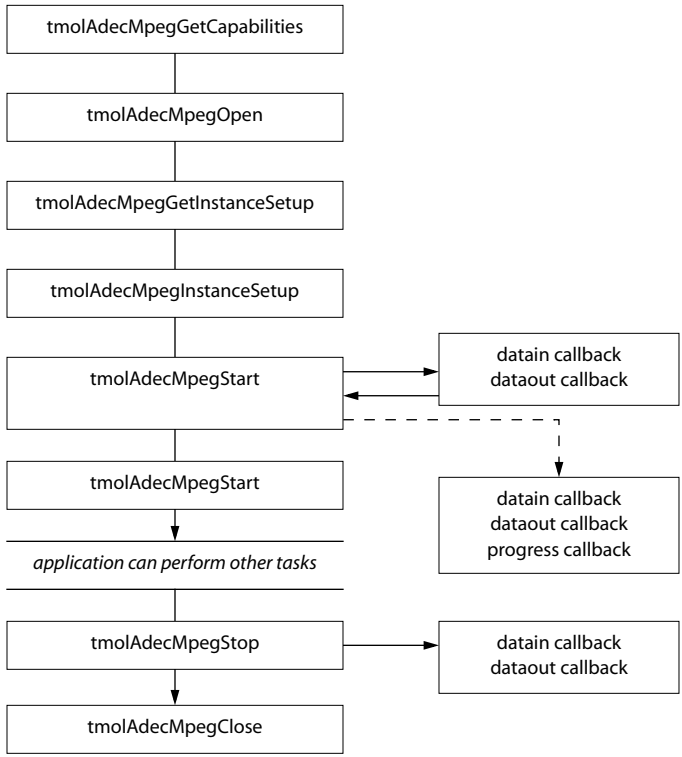


Figure 23 OL Layer Data Streaming Flow Control

Callback Function Requirements

The following table indicates the mandatory and optional callback functions used by the MPEG audio decoder.

Callback Function	Use
datainFunc (mandatory)	Used for data streaming to obtain full packets containing coded audio data. The tsaDefaults library provides a default function automatically.
dataoutFunc (mandatory)	Used for data streaming to obtain empty packets where decoded audio data will be stored. The tsaDefaults library provides a default function automatically.
controlFunc (mandatory)	Used to pass configuration command to the decoder. The tsaDefaults library provides a default function automatically.
progressFunc (mandatory)	Used by the decoder to report the decoders progress to the application. The tsaDefaults library provides a default function automatically.

MPEG Audio Decoder Data Structures

This section presents the TriMedia MPEG-1 Layer II and Layer III audio decoder data structures.

tmolAdecMpegCapabilities_t

```
typedef struct tmolAdecMpegCapabilities_t {
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmolAdecMpegCapabilities_t, *ptmolAdecMpegCapabilities_t;
```

Description

Standard TSSA capabilities structure. Used by applications to find out about the inputs and outputs of the component.

tmAdecMpegProgressFlags_t

```
typedef enum {
   ADEC_MPEG1_PROG_REPORT_FORMAT       = 0x01,
   ADEC_MPEG1_PROG_REPORT_FIND_SYNC    = 0x02,
   ADEC_MPEG1_PROG_REPORT_EVERY_FRAME  = 0x04
} tmAdecMpegProgressFlags_t;
```

Description

Controls the operation of the progress callback function. An application programmer can request notification in any of these cases. These flags are used to configure the progress function behavior during instance setup. In addition to that they are also used during the data streaming. Whenever the library calls the progress function, it indicates via the in progressCode field of the progress arguments which progress flag caused the function call.

tmAdecMpegMode_t

```
typedef enum {
   ADEC_MPEG1_STEREO          = 0x00000001,
   ADEC_MPEG1_JOINT_STEREO    = 0x00000002,
   ADEC_MPEG1_DUAL_CHANNEL    = 0x00000004,
   ADEC_MPEG1_SINGLE_CHANNEL  = 0x00000008
} tmAdecMpegMode_t;
```

Description

Describes the mode of the encoded audio. This type is used in the structure **tmAdecMpegFormat_t**.

tmAdecMpegLayer_t

```
typedef enum {
   ADEC_MPEG1_LAYER1  = 0x01,
   ADEC_MPEG1_LAYER2  = 0x02,
   ADEC_MPEG1_LAYER3  = 0x03
} tmAdecMpegLayer_t;
```

Description

Describes the encoding mode of the current stream. Reported in the **tmAdecMpegFormat_t** structure, as found in the bit stream.

tmAdecMpegCopyright_t

```
typedef enum {
   ADEC_MPEG1_COPYRIGHT_ON   = 0x01,
   ADEC_MPEG1_COPYRIGHT_OFF  = 0x02
} tmAdecMpegCopyright_t;
```

Description

Describes the copyright state of the current stream. Reported in the **tmAdecMpeg-Format_t** structure, as found in the bit stream.

tmAdecMpegOriginal_t

```
typedef enum {
   ADEC_MPEG1_ORIGINAL  = 0x01,
   ADEC_MPEG1_COPY      = 0x02
} tmAdecMpegOriginal_t;
```

Description

Describes the state of the "original" bit in the current stream. Reported in the **tmAdec-MpegFormat_t** structure, as found in the bit stream.

tmAdecMpegProtection_t

```
typedef enum {
   ADEC_MPEG1_CRC_ON   = 0x01,
   ADEC_MPEG1_CRC_OFF  = 0x00
} tmAdecMpegProtection_t;
```

Description

Tells whether or not CRC checksum are used to protect the transmitted bit stream. Reported in the **tmAdecMpegFormat_t** structure, as found in the bit stream.

tmAdecMpegPrivate_t

```
typedef enum {
   ADEC_MPEG1_PRIVATE_ON   = 0x01,
   ADEC_MPEG1_PRIVATE_OFF  = 0x02
} tmAdecMpegPrivate_t;
```

Description

Describes the state of the "private" bit in the current stream. Reported in the **tmAdecMpegFormat_t** structure, as found in the bit stream.

tmAdecMpegEmphasis_t

```
typedef enum {
   ADEC_MPEG1_NO_EMPHASIS     = 0x01,
   ADEC_MPEG1_50_15_EMPHASIS  = 0x02,
   ADEC_MPEG1_CCITT_EMPHASIS  = 0x03,
} tmAdecMpegEmphasis_t;
```

Description

Tells a user whether or not emphasis has been applied to the current stream. Reported in the **tmAdecMpegFormat_t** structure, as found in the bit stream.

tmAdecMpegSecOutputMode_t

```
typedef enum {
   ADEC_MPEG1_SEC_OUT_DISABLED  = 0x01,
   ADEC_MPEG1_SEC_OUT_1937      = 0x02,
} tmAdecMpegSecOutputMode_t;
```

Description

Sets the mode of operation for the second audio output. Always **ADEC_MPEG1_SEC_OUT_DISABLED** in this release.

tmolAdecMpegInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t   defaultSetup;
    tmAdecMpegSecOutputMode_t   secondOutputMode;
} tmolAdecMpegInstanceSetup_t, *ptmolAdecMpegInstanceSetup_t;
```

Fields

`defaultSetup`	Pointer to the default instance setup struct, refer to tsa.h.
`secondOutputMode`	To allow for 1937 output. Must be **ADEC_MPEG1_SEC_OUT_DISABLED** in this release.

Description

Configure the component for operation. Standard TSSA callback functions can be provided.

tmAdecMpegFormat_t

```
typedef struct AdecMpegFormat_t {
   tmAdecMpegLayer_t        layer;
   tmAdecMpegMode_t         eMode;
   UInt32                   bitRate;
   tmAdecMpegCopyright_t    copyright;
   tmAdecMpegProtection_t   protection;
   tmAdecMpegPrivate_t      private;
   tmAdecMpegOriginal_t     original;
   tmAdecMpegEmphasis_t     emphasis;
   Float                    sampleRate;
} tmAdecMpegFormat_t;
```

Fields

`layer`	Encoding method, layer 1, 2, or 3.
`emode`	Stereo mode.
`bitRate`	Encoded bit rate.
`copyright`	Recovered from bit stream.
`protection`	Is CRC used? Recovered from bit stream.
`private`	Recovered from bit stream.
`original`	Recovered from bit stream.
`emphasis`	Recovered from bit stream.
`sampleRate`	Recovered from bit stream.

Description

A structure of this type is passed to progress function when the sync word is found in a bit stream. An application can use this to determine the nature of the stream.

MPEG Audio Decoder Functions

This section presents the TriMedia MPEG-1 Layer II audio decoder functions.

tmolAdecMpegGetCapabilities

```
extern tmLibappErr_t tmolAdecMpegGetCapabilities (
   ptmolAdecMpegCapabilities_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 can be used to determine the capabilities of the audio decoder.

tmolAdecMpegOpen

```
extern tmLibappErr_t tmolAdecMpegOpen (
   Int   *instance
);
```

Parameters

instance	Pointer (returned) to the instance .

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_MEMALLOC_FAILED	Memory could not be allocated for the internal variables.

Description

Creates an instance of a MPEG audio decoder, and sets the instance variable to point to the audio decoder instance. Allocates memory for the instance variable.

tmolAdecMpegClose

```
extern tmLibappErr_t tmolAdecMpegClose (
   Int   instance
);
```

Parameters

`instance`	Instance value, as returned by **tmxlAdecMpegOpen**.

Return Codes

`TMLIBAPP_OK`	Success
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not open.

Description

This function will shut down an instance of the decoder. The instance must have been stopped prior to calling the function.

tmolAdecMpegGetInstanceSetup

```
extern tmLibappErr_t tmolAdecMpegGetInstanceSetup(
   Int                              instance,
   ptmolAdecMpegInstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance, as returned by **tmolAdecMpegOpen**.
`setup`	Pointer to a variable in which to return a pointer to setup data.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not open.

Description

The **tmolAdecMpegGetInstanceSetup** function is used to return a pointer to the decoders default OL Layer instance setup structure. The decoder creates this structure when the component is opened. After obtaining the pointer to the structure, the application can initialize specific instance values before calling **tmolAdecMpegInstanceSetup**.

tmolAdecMpegInstanceSetup

```
extern tmLibappErr_t tmolAdecMpegInstanceSetup (
   Int                              instance,
   ptmolAdecMpegInstanceSetup_t     setup
);
```

Parameters

`instance`	Instance, as returned by **tmalAdecMpegOpen**.
`setup`	Pointer to the setup structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The desired instance is not open.
`TMLIBAPP_ERR_NULL_PROGRESSFUNC`	The progress function callback pointer is Null.
`TMLIBAPP_ERR_NULL_DATAINFUNC`	The datain function callback pointer is Null.
`TMLIBAPP_ERR_NULL_DATAOUTFUNC`	The dataout function callback pointer is Null.
`TMLIBAPP_ERR_NULL_CONTROLFUNC`	The control function callback pointer is Null.
`TMLIBAPP_ERR_UNSUPPORTED_DATACLASS`	The input/output dataClass is not **avdcAudio**.
`TMLIBAPP_ERR_UNSUPPORTED_DATATYPE`	The input dataType is not **atfMPEG** or the output dataType is not **atfLinearPCM**.
`TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE`	The input dataSubtype is not either **amfMPEG_Layer1**, **amfMPEG_Layer2** or **amfMPEG_Layer3**, or the output data subtype is not **apfStereo16**.
`TMLIBAPP_ERR_NULL_IODESC`	Can assert if the input descriptor is Null.
`TMLIBAPP_ERR_NO_QUEUE`	The output descriptor has no full.

Description

This function configures the decoder.

tmolAdecMpegInstanceConfig

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

Parameters

`instance`	Instance, as returned by **tmolAdecMpegOpen**.
`flags`	Not used.
`args`	Pointer to the configuration arguments.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The desired instance is not open.
`TMLIBAPP_ERR_INVALID_COMMAND`	The configuration command is not recognized.

Description

This function can be used to change instance parameters after the component has been initialized and during data streaming operation. At present, no commands are implemented; this might change in the future.

tmolAdecMpegStart

```
extern tmLibappErr_t tmolAdecMpegStart (
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmalAdecMpegOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not open or set up.

Description

This function begins data streaming for the decoder. At the AL layer, it invokes a function that is an infinite loop. At the OL layer, this loop is spawned as a task.

tmolAdecMpegStop

```
extern tmLibappErr_t tmolAdecMpegStop (
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmxlAdecMpegOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not open or setup.

Description

This function stops the audio decoder from streaming data.

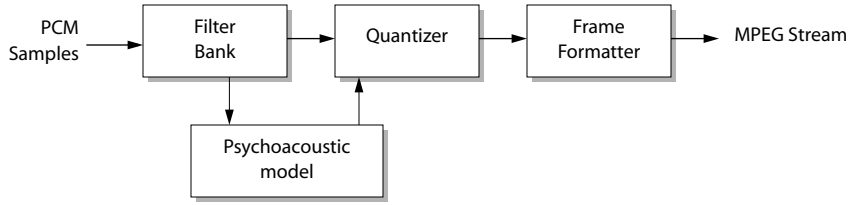
Chapter 15

MPEG-1 Audio Encoder (AencMpeg) API

Introduction

The MPEG-1 audio encoder library can be used to encode ISO 11172-3 compatible audio bit streams from 16-bit PCM data. This version of the library supports Layer II (Layer III is available on request). In Layer II mode the library can be used at both the AL and the OL layer. In Layer III mode, however, only the streaming mode interface is supported.

MPEG1 uses lossy sub-band coding techniques to achieve high compression rates. The PCM samples are transformed to the frequency domain by a polyphase filter bank. A psychoacoustic model calculates thresholds to control the quantizer. Finally additional information is added to build the MPEG stream.



Supported MPEG Modes

The encoder supports the following modes:

Domain	Modes
channels	(mono)/stereo
bit rates	32, 48, 56, 64, 80, 96, 112, 128, 160, 192 kb/s per channel
sample rates	32, 44.1, 48 kHz
error protection	optional CRC
flags	copyright, original

Emphasis and mono encoding are currently not supported.

Comparison of MPEG Audio Layers II and III

This table shows minimum values for the delay by the algorithms themselves. The delay may increase significantly if large output buffers are used.

Domain	Layer II	Layer III
Bit rates (per channel)	32-384 kb/s	32-320kbits/s

Domain	Layer II	Layer III
transparent at	128 kb/s	64 kb/s
min. delay	35 ms	59 ms
features	1024-point FFT	MDCT Higher frequency resolution Huffman-coding Variable bitrate

AencMpeg1 Inputs and Outputs

The MPEG-1 audio encoder library supports one input and one output pin. At its input pin, it expects TSSA audio packets with linear PCM data in **apfStereo16** format (or **apfMono16** in future versions). This version of the library supports no other format. The encoder supports the following sampling frequencies: 32000.0 Hz, 44100.0 Hz, and 48000.0 Hz. You can install the input data format either during instance setup, in the input descriptor of the default instance setup variable, or later on, by just sending a packet with the proper format to the encoder library.

The output format, and therefore the MPEG layer, is determined during the execution of **tmXlAencMpeg1InstanceSetup**. There are two ways to choose the mode in which the encoder works:

- The output descriptor of the default instance setup variable contains an output format, which is either **amfMPEG1_Layer2** or **amfMPEG1_layer3**.
- The output descriptor of the default instance setup variable has a null pointer in its format field. In this case, the value of the layer field of the instance setup variable determines the MPEG layer (and therefore the output format). A proper format is installed accordingly by the library.

The MPEG-1 audio encoder does not attach time stamps to the MPEG output packets. This must be done by a component operating downstream.

Run Time Behavior

The MPEG-1 encoding library can be used in both non-streaming and streaming mode. In the non-streaming mode, the application using the library must implement the necessary buffer management of the PCM data and compressed MPEG audio data. The library exposes one processing function for use in non-streaming mode: **tmalAencMpeg1-EncodeFrame**. The application must ensure that the encoder is always provided with a buffer of 1152 PCM samples and with a sufficiently large empty buffer for the encoded data when this function is called.

In streaming mode, buffer management is implemented by the encoder library. The component sending PCM data packets to the encoder can send any amount of samples at a time. The only restriction is that the last sample of the packet is complete and not

split across packet boundaries. At the output side, the encoder fills the empty output packets entirely before it sends them to the full queue. If the packets are smaller than a compressed audio frame, the encoder sends multiple packets per frame. If they are larger, the encoder puts multiple frames into one packet. Note that the choice of packet size during system setup affects the system performance and the system latency: larger packets account for a lower processor load but for a longer encoding delay.

When operated in streaming mode, the encoder can be forced to send out a partially filled output packet at any time by using the **tmolAencMpeg1InstanceConfig** function with the command **AENC_MPEG1_CONFIG_FLUSH_OUTPUT**. Note that the execution of this command might be delayed because the command is sent to the encoder component through an operating system message queue.

The example program **exolAencMpeg1** shows how the encoder can be used in a real-time encoding application. This example samples audio input with the audio digitizer. The samples are sent to the MPEG-1 audio encoder. The program supports two different output modes. In the first mode, the encoder is connected to the File Writer component which stores the encoded audio bit stream in a file. In the second mode, the MPEG-1 audio decoder receives the encoder output and decodes it. Then, the PCM samples are played with the audio renderer.

Performance

The processor resources requirements have been measured with an OL layer encoder application.

The performance measure unit is MIPS which is independent of the TriMedia clock frequency. A 100 MHz TriMedia executes 100 million VLIW instructions and therefore provides 100 MIPS. Note that the processor load largely depends on the complexity of the audio material to be encoded.

Sample Rate/ Bit Rate	Stereo	Mono
44100 Hz / 224 kb/s	38 MIPS	20 MIPS

AencMpeg1 Errors

The following component-specific error messages can be reported by the error function of the MPEG-1 audio encoder, when operated in streaming mode:

`AENC_MPEG1_ERR_ILL_SAMPLERATE` A new input format is installed and the sampling rate is neither 32 kHz, 44.1 kHz, nor 48 kHz.

`AENC_MPEG1_ERR_ILL_DATASIZE` Incomplete samples are received in one of the input data packets. The **dataSize** field of the TSSA input packets must contain a multiple of 4 bytes in **apfStereo16** mode and a multiple of 2 bytes in **apfMono16** mode.

If a new input format is installed and the data class is not **avdcAudio** or the data type is not **atfLinearPCM** or the data subtype is not **apfStereo16**, one of the following standard errors is reported by the error function:

```
TMLIBAPP_ERR_UNSUPPORTED_DATACLASS
TMLIBAPP_ERR_UNSUPPORTED_DATATYPE
TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE
```

In non-streaming mode, the function **tmalAencMpeg1EncodeFrame** is called to encode MPEG-1 layer 2 audio frames. When calling this function, ensure that the input packet contains exactly one frame of PCM samples (1152 samples) and that the output packet is large enough to store the compressed frame. If these conditions are not met, the encoder returns one of these the error messages:

```
AENC_MPEG1_ERR_NOT_ENOUGH_INPUT_SAMPLES
AENC_MPEG1_ERR_OUTBUF_TOO_SMALL
```

AencMpeg1 Progress

If the progress flag

```
AENC_MPEG1_PROG_REPORT_EVERY_FRAME
```

is installed during instance setup, the progress function of the MPEG-1 audio encoder is called whenever the encoding of an audio frame (1152 samples) is completed. The progress arguments do not contain any specific information. The progress function can be used to measure the processor load of the audio encoder or just to count the number of encoded frames.

AencMpeg1 Configuration

The function **tmXlAencMpeg1InstanceConfig** can influence the behavior of the encoder while it is running. The current version of the library supports one command:

```
AENC_MPEG1_CONFIG_FLUSH_OUTPUT
```

When this command is sent to the encoder, its current output packet is sent out, no matter how much it is filled. This feature might be useful to decrease the system latency.

Audio Encoder Data Structures

This section presents the TriMedia MPEG-1 Layer II and Layer III audio encoder data structures.

tmalAencMpeg1ConfigTypes_t

```
typedef enum {
   AENC_MPEG1_CONFIG_FLUSH_OUTPUT = tsaCmdUserBase + 0x00
} tmalAencMpeg1ConfigTypes_t;
```

Fields

AENC_MPEG1_CONFIG_FLUSH_OUTPUT	Flush the current output packet, no matter how much it is filled.

Description

Enumerates the commands recognized by **tmalAencMpeg1InstanceConfig**.

AENC_MPEG1_CONFIG_FLUSH_OUTPUT can be used to flush the current output packet no matter how much it is filled.

tmalAencMpeg1Layer_t

```
typedef enum {
   AENC_MPEG1_LAYER1   = 0x01,
   AENC_MPEG1_LAYER2   = 0x02,
   AENC_MPEG1_LAYER3   = 0x03
} tmalAencMpeg1Layer_t;
```

Description

Enumerates MPEG-1 layers.

tmalAencMpeg1Copyright_t

```
typedef enum {
   AENC_MPEG1_COPYRIGHT_ON   = 0x01,
   AENC_MPEG1_COPYRIGHT_OFF  = 0x02
} tmalAencMpeg1Copyright_t;
```

Description

Enumerates copyright values. A bitstream can be marked as copyright-protected.

tmalAencMpeg1Protection_t

```
typedef enum {
   AENC_MPEG1_CRC_ON   = 0x01,
   AENC_MPEG1_CRC_OFF  = 0x00
} tmalAencMpeg1Protection_t;
```

Description

Enumerates protection values. During instance setup, you can determine whether the encoder puts a cyclic redundancy check (CRC) in every encoded MPEG-1 audio frame.

tmalAencMpeg1Private_t

```
typedef enum {
   AENC_MPEG1_PRIVATE_ON   = 0x01,
   AENC_MPEG1_PRIVATE_OFF  = 0x02
} tmalAencMpeg1Private_t;
```

Description

Enumerates privacy values. During instance setup, you can determine whether the private bit in the headers of the encoded audio frames is set.

tmalAencMpeg1Original_t

```
typedef enum {
   AENC_MPEG1_ORIGINAL   = 0x01,
   AENC_MPEG1_COPY       = 0x02
} tmalAencMpeg1Original_t;
```

Description

During instance setup, you can determine whether the encoder marks the MPEG bitstream as original or as copy.

tmalAencMpeg1Emphasis_t

```
typedef enum {
   AENC_MPEG1_NO_EMPHASIS     = 0x01,
   AENC_MPEG1_50_15_EMPHASIS  = 0x02,
   AENC_MPEG1_CCITT_EMPHASIS  = 0x03,
} tmalAencMpeg1Emphasis_t;
```

Description

Determines the type of pre-emphasis applied to the input audio signal before it enters the encoder.

tmalAencMpeg1Capabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmalAencMpeg1Capabilities_t, *ptmalAencMpeg1Capabilities_t;
```

Fields

`defaultCapabilities`	Pointer to a default capabilities structure. (Refer to tsa.h.)

Description

This structure contains a description of the capabilities of the MPEG-1 audio encoder library. It is filled by the **tmalAencMpeg1GetCapabilities** function.

tmAencMpeg1ProgressFlags_t

```
typedef enum {
   AENC_MPEG1_PROG_REPORT_EVERY_FRAME  = Øx01
} tmAencMpeg1ProgressFlags_t;
```

Description

Enumerates MPEG progress flags.

tmalAencMpeg1InstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   tmalAencMpeg1Layer_t        layer;
   UInt32                      bitRate;
   UInt32                      quality;
   tmalAencMpeg1Copyright_t    copyright;
   tmalAencMpeg1Protection_t   protection;
   tmalAencMpeg1Private_t      private;
   tmalAencMpeg1Original_t     original;
   tmalAencMpeg1Emphasis_t     emphasis;
} tmalAencMpeg1InstanceSetup_t, *ptmalAencMpeg1InstanceSetup_t;
```

Fields

`defaultSetup`	Pointer to the default instance setup. (Refer to tsa.h.)
`layer`	Determines the encoding profile. Note that the MPEG layer can also be determined by the format of the output descriptor. If the format is installed, it overrides the value in this structure. Layer I is currently not supported. (Refer to **tmalAencMpeg1Layer_t**.)
`bitrate`	Determines the rate of the encoder output (kilobits per second). Legal values are 32, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 384 for layer II and 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320 for layer III.
`quality`	Used only in layer III mode, this determines the break-up condition for the bit allocation iteration loop. A lower number stands for more iterations and better audio quality. Supported values are integers from 0 to 30.
`copyright`	Determines whether the MPEG stream will be marked as copyright protected. (Refer to **tmalAencMpeg1Copyright_t**.)
`protection`	Determines whether a CRC word is calculated and inserted into the MPEG bitstream for every audio frame. (Refer to **tmalAencMpeg1Protection_t**.)
`private`	Determines the value of the privacy bit to be written into the MPEG frame headers. (Refer to **tmalAencMpeg1Private_t**.)
`original`	Determines whether the MPEG bit stream is marked as original. (Refer to **tmalAencMpeg1Original_t**.)

`emphasis`	Characterizes the nature of the pre-emphasis applied to the audio input outside of the encoder. Currently no emphasis is used. (Refer to **tmalAencMpeg1Emphasis_t**.)

Description

A structure of this type configures the MPEG-1 audio encoder when operated at the AL layer. For the OL layer, a similar structure is used. A pointer to a struct of this type is passed to **tmalAencMpeg1InstanceSetup** as an argument. A pre-configured "template" of this structure can be obtained by **tmalAencMpeg1GetInstanceSetup**.

Audio Encoder Functions

This section presents the TriMedia MPEG-1 Layer II and Layer III audio encoder functions.

tmolAencMpeg1GetCapabilities

```
tmLibappErr_t tmolAencMpeg1GetCapabilities (
   ptmolAencMpeg1Capabilities_t   *pcap
);
```

tmalAencMpeg1GetCapabilities

```
tmLibappErr_t tmolAencMpeg1GetCapabilities (
   ptmalAencMpeg1Capabilities_t   *pcap
);
```

Parameters

`pcap`	Pointer to a variable in which to return a pointer to capabilities data.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Fills in the pointer of a static **tmolAencMpeg1Capabilities_t** structure maintained by the encoder to describe the capabilities and requirements of this library.

tmolAencMpeg1Open

```
tmLibappErr_t tmolAencMpeg1Open (
   Int   *instance
);
```

tmalAencMpeg1Open

```
tmLibappErr_t tmalAencMpeg1Open (
   Int   *instance
);
```

Parameters

`instance`	Pointer (returned) to an encoder instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	A **memalloc** failed.
`TMLIBAPP_ERR_NO_INSTANCE_AVAILABLE`	No further layer instance is available.

Side Effects

Creates an instance of an encoder and **calloc**'s an instance structure. Allocates an instance setup structure and fills it with default values.

tmolAencMpeg1Close

```
extern tmLibappErr_t tmolAencMpeg1Close (
   Int   instance
);
```

tmalAencMpeg1Close

```
extern tmLibappErr_t tmalAencMpeg1Close (
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by an 'open' function.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is invalid.

Description

Closes this instance of the encoder and deletes task. Frees instance variable memory and setup variable memory.

tmolAencMpeg1GetInstanceSetup

```
extern tmLibappErr_t tmolAencMpeg1GetInstanceSetup (
   Int                              instance,
   ptmolAencMpeg1InstanceSetup_t  *setup
);
```

tmalAencMpeg1GetInstanceSetup

```
extern tmLibappErr_t tmalAencMpeg1GetInstanceSetup (
   Int                              instance,
   ptmalAencMpeg1InstanceSetup_t  *setup
);
```

Parameters

`instance`	Instance, as returned by an 'open' function.
`setup`	Pointing to variable in which to return a pointer to setup data.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is invalid.

Description

Returns (1) a pointer to the setup structure allocated by the open function or (2) the current setup structure after the setup function has been called.

tmolAencMpeg1InstanceSetup

```
extern tmLibappErr_t tmolAencMpeg1InstanceSetup (
   Int                              instance,
   tmolAencMpeg1InstanceSetup_t   *setup
);
```

tmalAencMpeg1InstanceSetup

```
extern tmLibappErr_t tmalAencMpeg1InstanceSetup (
   Int                              instance,
   tmalAencMpeg1InstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance, as returned by an 'open' function.
`setup`	Pointer to the setup structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is invalid.
`TMLIBAPP_ERR_NO_QUEUE`	The queues for either the input pin or the first output pin are not assigned.
`AENC_MPEG1_ERR_LAYER_NOT_SUPPORTED`	The selected MPEG layer is not supported.

`TMLIBAPP_ERR_UNSUPPORTED_DATACLASS`
`TMLIBAPP_ERR_UNSUPPORTED_DATATYPE`
`TMLIBAPP_ERR_UNSUPPORTED_DATASUBTYPE`
	Incorrect format in the input or output descriptor.

`AENC_MPEG1_ERR_ILL_SAMPLERATE`	Unsupported sample rate specified in the input format.
`AENC_MPEG1_ERR_ILL_BITRATE`	Unsupported bit rate specified in the setup struct.
`AENC_MPEG1_ERR_ILL_QUALITY`	Unsupported quality value specified in the setup struct.
`AENC_MPEG1_ERR_ILL_EMPHASIS`	Unsupported emphasis value of the setup struct.

Description

Initializes the instance of the encoder and configures it.

tmolAencMpeg1Start

```
extern tmLibappErr_t tmolAencMpeg1Start (
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by an 'open' function.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The encoder has not been opened by this instance.
`TMLIBAPP_ERR_NOT_SETUP`	The encoder has not been initialized.

Description

Starts the encoder's **tmalAencMpeg1Start** function as a task.

tmalAencMpeg1Start

```
extern tmLibappErr_t tmalAencMpeg1Start (
   Int   instance
);
```

Parameters

instance	Instance, as returned by an 'open' function.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	The encoder has not been opened by this instance.
TMLIBAPP_ERR_NOT_SETUP	The encoder has not been initialized.

Description

Starts the data processing loop of the MPEG-1 encoder.

tmolAencMpeg1InstanceConfig

```
extern tmLibappErr_t tmolAencMpeg1InstanceConfig (
   Int                instance,
   Int32              flags,
   tsaControlArgs_t   args
);
```

Parameters

`instance`	Instance, as returned by an 'open' function.
`flags`	Flags used for the pSOS command queue which sends the command to the AL layer config function.
`args`	A structure containing the command, a parameter pointer, and a timeout value.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The encoder has not been opened by this instance.
`TMLIBAPP_ERR_NOT_SETUP`	The encoder has not been initialized.

The function can also return error messages from the command queue handler.

Description

Invokes the default stop procedure which stops the encoder task and sends pause packets to the connected components.

tmalAencMpeg1InstanceConfig

```
extern tmLibappErr_t tmalAencMpeg1InstanceConfig   (
   Int                 instance,
   ptsaControlArgs_t   cmdArgs
);
```

Parameters

`instance`	Instance, as returned by an 'open' function.
`cmdArgs`	A structure containing the command and a parameter pointer. This function does not use its timeout field nor does it get values.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The encoder has not been opened by this instance.
`TMLIBAPP_ERR_NOT_SETUP`	The encoder has not been initialized.
`TMLIBAPP_ERR_INVALID_COMMAND`	The config function cannot interpret the command.

Description

Issues a command to the encoder. This function can be used when the encoder is running.

tmolAencMpeg1Stop

```
extern tmLibappErr_t tmolAencMpeg1Stop (
   Int   instance
);
```

tmalAencMpeg1Stop

```
extern tmLibappErr_t tmalAencMpeg1Stop (
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by an 'open' function.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The encoder has not been opened by this instance.
`TMLIBAPP_ERR_NOT_SETUP`	The encoder has not been initialized.

Description

The OL layer function invokes the default stop procedure which stops the encoder task and sends pause packets to the connected components.

The AL layer function forces the encoder to the main processing loop of the start function.

tmalAencMpeg1EncodeFrame

```
extern tmLibappErr_t tmalAencMpeg1EncodeFrame (
   Int             instance,
   tmAvPacket_t   *inpacket,
   tmAvPacket_t   *outPacket
);
```

Parameters

`instance`	Instance, as returned by an 'open' function.
`inpacket`	Pointer to a full packet carrying PCM stereo data.
`outPacket`	Pointer to an empty packet into which the encoder writes the encoded audio frame.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The encoder has not been opened by this instance
`TMLIBAPP_ERR_NOT_SETUP`	The encoder has not been initialized.
`AENC_MPEG1_ERR_NOT_ENOUGH_INPUT_SAMPLES`	The input packet contains less samples than a full audio frame (1152 for layer II).
`AENC_MPEG1_ERR_OUTBUF_TOO_SMALL`	The empty output packet is not large enough to store an encoded audio frame.
`AENC_MPEG1_ERR_LAYER_NOT_SUPPORTED`	The encoder is not supporting the layer chosen during instance setup in non streaming mode.

Description

Encodes one frame of audio data. The user of this function must ensure that the input packet contains the exact number of samples required for one frame. The encoder does not perform any type of buffering between subsequent calls of this function.

Note that this function is only applicable for encoding Layer II bit streams! It is also important to mention that padding bytes are not inserted when this function is used. In future versions of the library this will be done.

