ANALOG DEVICES

AAC DECODER DEVELOPER'S GUIDE

ANALOG DEVICES, INC.

www.analog.com

AUTHOR: ADA

KT-41 (REV. 1.02, 2007-10-08)

Table of Contents

List of Figures

List of Tables

List of Code Examples

Copyright, Disclaimer & Trademark Statements

Copyright Information

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Supply of this Implementation of AAC technology does not convey a license nor imply any right to use this Implementation in any finished end-user or ready-to-use final product. An independent license for such use is required

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, SHARC, TigerSHARC, CROSSCORE, VisualDSP, VisualDSP++, VisualAudio and EZ-KIT Lite are registered trademarks "®" of Analog Devices, Inc.

Collaborative™ is a trademark of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

1 Introduction

This document describes how to use the AAC decoder family of libraries.

1.1 Scope

This guide is intended for C-language developers proficient in the application of the Analog Devices Blackfin family of DSPs and related software tools. It is assumed that the reader is familiar with all aspects of these DSPs.

1.2 Target Platform

The AAC decoder libraries are designed for the Analog Devices Blackfin DSPs.

1.3 Organisation of this Guide

Information supplied in this guide is organised in the following way:

- Section 2 provides the specifications of the AAC decoder.
- Section 3 is a "Quick Start Guide" about how to integrate the AAC decoder library.
- Section 4 describes each of the functions in the AAC decoder library.
- Section 5 describes linking information, memory usage and performance of the AAC decoder library.
- Appendix A: provides an example on how to use the AAC decoder library.

1.4 Version Information

Note that this document refers to the following software versions:

MPEG-4 AAC-LC Stereo Decoder: ADI Version 3.0.x

MPEG-4 (with DAB support) HE-AAC v2 Stereo Decoder: ADI Version 3.0.x

Any information that accompanies the software libraries (in the form of 'readme' files, etc.) should be assumed to supersede the information contained in this document.

1.5 System Requirements

This module was built and tested with VDSP++ 4.5 February 2007 update of the tools, versions of which are listed in Table 1-1. To ensure the maximum level of compatibility, please use these versions of the tools.

Tool Name	Tool Description	Tool Version
ccblkfn	C/C++ compiler	7.2.4.3

easmblkfn	Blackfin assembler	2.6.16.5
linker	Linker	3.8.0.4

Table 1-1: Tool revisions revision (as reported using the –version command line option)

1.6 Acronyms

AAC	Advanced Audio Coding
AAC-LC	Low Complexity AAC
ADI	Analog Devices Inc.
API	Application Program Interface
HE-AAC	High Efficiency AAC (usage of SBR with AAC-LC)
HE-AAC v2	Enhanced High Efficiency AAC (usage of SBR and PS tools with AAC-LC)
MPEG	Moving Pictures Experts Group
PNS	Perceptual Noise Substitution
PS	Parametric Stereo
SBR	Spectral Band Replication

1.7 References

[1] International Standard ISO/IEC 14496-3:2005(E) "Information technology - Coding of audio, picture, multimedia and hypermedia information – Part 3: Audio", ISO/IEC JTC1/SC29 (MPEG-4), 2005

[2] International Standard ISO/IEC 14496-3:2005/Amd.2:2006(E) "Information technology - Coding of audio, picture, multimedia and hypermedia information – Part 3: Audio – AMENDMENT 2: Audio Lossless Coding (ALS), new audio profiles and BSAC extensions", ISO/IEC JTC1/SC29 (MPEG-4), 2006

[3] International Standard ISO/IEC 14496-4:2004(E) "Information technology - Coding of audio, picture, multimedia and hypermedia information – Part 4: Conformance testing", ISO/IEC JTC1/SC29 (MPEG-4), 2004

[4] International Standard ISO/IEC 14496-4:2004/Amd.8:2005(E) "Information technology - Coding of audio, picture, multimedia and hypermedia information – Part 4: Conformance testing, Amendment 8 High Efficiency Advanced Audio Coding, audio BIFS, and structured audio conformance", ISO/IEC JTC1/SC29 (MPEG-4), 2005

[5] International Standard ISO/IEC 14496-4:2004/Amd.11:2006(E) "Information technology – Coding of audio, picture, multimedia and hypermedia information – Part 4: Conformance testing – Amendment 11 Parametric stereo conformance", ISO/IEC JTC1/SC29 (MPEG-4), 2006

[6] VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin processors, ADI, 2005.

[7] *Silicon anomaly workaround guide for developers*, Analog Devices Inc., KT-68, 2007.

[8] *MPEG-4 AAC-LC Stereo Decoder BF Spec Sheet*, Analog Devices Inc., KT-286 2007.

[9] *MPEG-4 HE-AAC v2 Stereo Decoder BF Spec Sheet*, Analog Devices Inc., KT-312, 2007.

1.8 Additional Information

For more information on the latest ADI processors, silicon errata, code examples, development tools, system services and device drivers, technical support and any other additional information, please visit our website at www.analog.com/processors.

2 Specifications

The Analog Devices AAC decoder library provides the functionality of decoding AAC-LC bitstreams. The library was designed for the Blackfin 53x family of DSPs.

2.1 Decoder features

All AAC-LC and HE-AAC decoders support the following:

- MPEG-4 AAC-LC audio object type
- Core AAC-LC frame block length of 1024 samples
- All sampling frequencies (up to 96 kHz) as specified by [1] (ISO/IEC 14496-3, a.k.a. 'MPEG-4 AAC-LC')
- All bit rates including Variable Bit Rates (VBR) of the AAC standards
- Supports the following types of input bitstreams: RAW, ADIF, ADTS.
- Fully re-entrant and multi-instancing capable

All Stereo decoders support the following:

- Supports up to two output audio channels (Stereo)
- Supports decoding of front left and front right channels from multi-channel audio bitstreams.

2.1.1 HE-AAC v2 Stereo Decoder (with DAB support)

In addition to the common features listed above, the MPEG-4 HE-AAC v2 stereo decoder (with DAB support) features the following:

- Support for the MPEG-4 SBR audio object type as specified by [1]
- Support for the MPEG-4 PS audio object type as specified by [1]
- Alternative core AAC-LC frame block length of 960 samples

2.2 Input Format

The decoder supports AAC-LC bit streams as specified in ISO/IEC 14496-3.

2.3 Output Format

The output from the AAC decoder are 16-bit PCM audio samples. The user can configure the decoder to output each channel in separate buffers, or interleave them, with configurable sample to sample strides for each channel.

2.4 Conformance

The AAC decoder has passed the ISO conformance tests as outlined in [3], [4], and [5], and is conformant at the profile and level specification of MPEG4 as indicated by Table 2-1.

Variant of AAC Decoder	MPEG4 profile@level
AAC-LC Stereo	AAC@Level 2
HE-AAC v2 Stereo (with DAB support)	High Efficiency AAC v2@Level 3

Table 2-1: ISO Conformance Point for each Library Variant

3 Quick Start

The purpose of this section is to provide a brief description of the AAC decoder API. This is to aid developers to integrate the AAC decoder module into their application code with minimal effort.

1. Section 3.1 provides a step by step usage guide for the AAC decoder.
2. Section 3.2 describes how to create multiple instances of the AAC decoder.
3. Section 3.3 provides example application code to illustrate the usage of the AAC decoder.
4. Section 3.4 discusses some system integration issues when using the AAC decoder.

3.1 Step-by-step Guide

1. Include the appropriate library header in the application:

   AAC-LC Stereo library - *adi_aaclc_decoder.h*

   HE-AAC v2 Stereo library - *adi_heaacv2_decoder.h*

2. **Create the decoder instance.** Create the memory setup structure, point the elements to appropriate memory blocks, and call adi_aacd_create (). Macros are available for the minimum required memory block sizes.
3. **Determine the container format**. Fill the input buffer (or a separate buffer) in order to parse the container format if required, by calling the ADTS/ADIF helper function, or a separate container format parser.
4. **Copy audio configuration information.** If a parser was used to extract audio configuration information from the bitstream container, then this needs to be copied from the container parser to the decoder, by filling in the adi_aacd_config_t and adi_aacd_stream_config_t data structs correctly.
5. **Initialise the decoder instance**. Create the structure to hold the requested configuration and initialise it with the desired values. Call adi_aacd_init(). Check for a successful function return code before proceeding.
6. Fill input buffer if necessary. The user should supply enough data for decoding a frame – this will either be ADI_AACD_MIN_BITSTREAM_BUFFER_NUMBYTES or the number of bytes for a transport layer packet (assuming that a transport layer packet holds an integral number of audio frames).
7. **Decode the input buffer.** Call adi_aacd_decoder() with the input bitstream parameters and act on the return result.
8. Repeat steps 6 & 7 as required. See section 4 for more information about the API functions.
9. When the song or audio program has been completely decoded, then a new song or audio program can be decoded by going back to step 3. Since the AAC audio bitstream does not contain any marker signalling the end of a stream, it is up to the system to detect when the end of a song or program has been reached. For example, when the end of a file has been

reached. It may be necessary for the application to flush the buffer at the end of a song if the decoder reports more data is required to decode the remaining data but there is no more available from the data source.

3.2 Creating and using multiple instances

The decoder library supports creation and usage of multiple instances, running in concurrent systems, provided the following conditions are satisfied:

- Multiple decoder instances, running concurrently in separate threads, must be supplied with dedicated scratch memory. This scratch memory must be persistent for the duration of the call to **`adi_aacd_decoder()`**. That is, it should not be over-written or corrupted by another instance. It may be used for other purposes in between calls.

- Multiple decoder instances, running in the same thread sharing a common scratch memory pool. This can only be done provided that for a decoder instance, a call to **`adi_aacd_decoder()`** is not interrupted, by a call to this function for another instance.

Note: One instance is created for each AAC input stream, by calling the functions **`adi_aacd_create()`** and **`adi_aacd_init()`** once, followed by repeated calls to **`adi_aacd_decoder()`** for the duration of the stream.

However it is possible to process multiple AAC bit streams (in sequence, one after the other) using the instance that is created to process the first AAC bit stream in the sequence. In this scenario, for each subsequent bit stream to be processed, the call to **`adi_aacd_create()`** is not required and thus should be skipped. The application can simply call **`adi_aacd_init()`**, and then commence feeding the new bitstream to the decoder.

3.3 Example Code

For an example application that uses the AAC decoder, please refer to Appendix A AAC Decoder Usage Example Code.

3.4 System Integration

This section highlights some known system integration issues when integrating the AAC decoder library.

There are currently no known issues. The user should follow the integration instructions contained in Section 3.1 .

4 AAC Decoder Programmer's Reference

This section provides a detailed description of the AAC decoder API structures and functions.

4.1 Global constants

4.1.1 adi_aacd_module_info

```
extern const adi_aacd_module_info_t adi_aacd_module_info;
```

This constant global struct holds information about the library. See 4.3.1 for details of the struct.

4.2 API Functions

4.2.1 adi_aacd_create ()

Prototype

```
adi_aacd_t* adi_aacd_create(
     adi_aacd_mem_t *per_instance_mem_blocks);
```

Description

This function is to be used to create a single instance of the decoder. Memory is allocated for the instance using the memory blocks supplied.

Parameters

Name: **`per_instance_mem_blocks`**

Type: **`adi_aacd_mem_t*`**

Direction: Input

Description: Structure that contains the pointers to memory pools used to store decoder internal state and scratch information.

Return Value

The function returns a valid pointer upon successful creation of the instance, otherwise a NULL pointer is returned. The latter is returned if the input pointers to the memory pools themselves are NULL.

4.2.2 adi_aacd_get_default_config()

Prototype

```
void adi_aacd_get_default_config(
    adi_aacd_config_t *config)
```

Description

This function allows the application to commence with a known good set of configuration values and modify them to suit their application. The values that the codec inserts are detailed in the table below:

use_circular_bitstream_buffer	1;
circular_buffer_length	0;
circular_buffer_base	NULL;
frame_properties_ptr	NULL;
stream_properties_ptr	NULL;
output_chan_config	ADI_AACD_2_0_CHAN_CONFIG;
output_chan_stride	1 (for all values in the array)
decoding_mode	ADI_AACD_DECODING_MODE_AACLC or ADI_AACD_DECODING_MODE_HEAACv2 depending on the library variant
selected_program_id	-1
container_format	ADI_AACD_RAW;
container_info	
num_channels	2
sampling_frequency_index	3 (48000 Hz)
sampling_frequency	48000
sbr_sampling_frequency	96000
frame_length_flag	0 (1024 samples per frame)
channel_configuration	0
num_front_channel_elements	0
num_side_channel_elements	0
num_back_channel_elements	0
num_coupled_channel_elements	0
num_lfe_channel_elements	0
channel_element_id[]	0

Table 4-1 adi_aacd_get_default_config() values

Parameters

Name: `config`

Type: `adi_aacd_config_t *`

Direction: Input/output

Description: configuration structure to initialise

Return value: This is a void function that returns no value

4.2.3 adi_aacd_init ()

Prototype

```
adi_aacd_return_code_t adi_aacd_init (
     adi_aacd_t *instance_handle,
     adi_aacd_config_t *config_params);
```

Description

This function must be called to initialize the decoder instance created by the *adi_aacd_create()* function. Part of this initialization uses a set of configuration parameters passed in by the application via the pointer *config_params*. The definition of its type **adi_aacd_config_t** is given in section 4.3.5 .

This function checks all configuration parameters for validity. In the event any parameter is found to be invalid, it will be modified to the most appropriate valid value.

In all cases, the application is notified via the return value that one or more configuration parameters were invalid. In addition the function also fills the application supplied output structure with the actual configuration values used.

The only configuration variables that may be changed after the call to adi_aacd_init() are described in the reference for the function adi_aacd_reconfigure() in section 4.2.5 .

Parameters

Name: `instance_handle`

Type: `adi_aacd_t *`

Direction: Input/output

Description: The decoder instance.

Name: `config_params`

Type: `adi_aacd_config_t *`

Direction: Input

Description: Structure containing configuration parameters. The decoder modifies the values in this structure if they are invalid.

Return value: The function returns one of two error codes given below:

- ADI_AACD_SUCCESS: The decoder initialised without error
- ADI_AACD_INVALID_CONFIG_PARAMS: One of the requested configurations is invalid, and over-ridden by the decoder using a default value. The user may still go ahead with calling adi_aacd_decoder, but should check which parameter has been over-ridden.
- ADI_AACD_INVALID_POINTER: One of the function arguments or one of the following requested configuration pointers was set incorrectly to NULL
    - circular_buffer_base set to NULL when use_circular_bitstream_buffer is set to 1
    - frame_properties_ptr set to NULL
    - stream_properties_ptr set to NULL
- ADI_AACD_TERMINAL_FAILURE: One of the requested configurations is illegal and prevents the decoder being configured properly. Any calls to adi_aacd_decoder will also result in the return code until adi_aacd_init is called with a legal configuration. One of the following conditions will result in this return code:
    - circular_buffer_ < ADI_AACD_MIN_BITSTREAM_BUFFER_NUMBYTES, when use_circular_bitstream_buffer is set to 1
    - output_chan_stride[i] set to zero or negative

4.2.4 adi_aacd_resynch()

Prototype

```
void adi_aacd_resynch(
     adi_aacd_t *instance_handle);
```

Description

This function resets the decoder to a known state, typically used when fast-forward and rewind operations are supported. The decoder is initialized to the same state as if the function **adi_aacd_init()** were called. However, application configuration parameters will not be reset and the existing configuration parameters will be kept unchanged. This is done since reconfiguring application parameters is very unlikely when resynchronisation is required. In the event that it is required, it can be done by using the supplied function **adi_aacd_reconfigure()** before calling **adi_aacd_resynch()**. Note that after resynchronisation is performed, it is assumed the decode operation that follows is for the start of a new frame. In the case of ADTS framing, the decoder must be resynchronised at an ADTS frame header.

Parameters

Name: `instance_handle`

Type: `adi_aacd_t *`

Direction: Input/output

Description: The decoder instance to resynchronize.

Return value:

- None.

4.2.5 adi_aacd_reconfigure ()

Prototype

```
adi_aacd_return_code_t adi_aacd_reconfigure(
     adi_aacd_t *instance_handle,
     adi_aacd_config_item_t config_item,
     adi_aacd_config_t *config_params)
```

Description

This function changes the parameters of the decoder as it operates, by supplying the new value in the config_params structure. Call this function if a decoding parameter has been changed and the change is to be effected.

Below is a complete list of decoding parameters that may be changed between calls to adi_aac_decoder():

Configuration Item	Description
-	Currently no parameters can be reconfigured at run time

Parameters

Name: `instance_handle`

Type: `adi_aacd_t *`

Direction: Input/output

Description: The decoder instance.

Name: `config_item`

Type: `adi_aacd_config_item_t`

Direction: Input

Description:. Value indicating whether all (ADI_AACD_ALL_RECONFIG_PARAMS) or a single configuration item supplied in the third argument is to be reconfigured. Parameters which are indicated in this guide as being not reconfigurable will be ignored by this function.

Name: `config_params`

Type: `adi_aacd_config_t *`

Direction: Output

Description: This structure contains the value, or values, to be reconfigured, as indicated in the second argument config_item.

Return value: The function returns one of two error codes given below:

- ADI_AACD_SUCCESS: The decoder reconfigured the selected parameter without error
- ADI_AACD_INVALID_CONFIG_PARAMS: The requested reconfiguration is invalid
- ADI_AACD_INVALID_POINTER: One of the function arguments was set incorrectly to NULL

4.2.6 adi_aacd_decoder()

Prototype

```
adi_aacd_return_code_t adi_aacd_decoder(
     adi_aacd_t *instance_handle,
     int num_input_bytes,
     adi_aacd_bitstream_data_t *input_bitstream,
     adi_aacd_output_status_t *dec_output_status,
     adi_aacd_audio_data_t *output_pcm_channels[]);
```

Description

This function is the main decoder routine, producing PCM data for each decoded channel. The decoder is designed to be called with an arbitrary number of bytes in the input buffer, to decode and output an audio segment, otherwise letting the user know if not enough input data is available to the decoder.

4.2.6.1.1 Usage

The following usage conditions must be satisfied for the correct usage of the decoder:

1. For every frame to be decoded, the first time the decoder is called (after calling **`adi_aacd_init()`** or **`adi_aacd_resynch()`**) with some input data, *input_bitstream* must be pointing to the beginning of the frame. Through mechanisms not defined in this document, it is the user's responsibility to determine the start of the frame.

   Note *input_bitstream* should not be confused with the base address of the application buffer that is passed as part of the configuration parameters in the function adi_aacd_init(). This input_bitstream shall always be pointing to the next byte to be consumed by the decoder.

2. If the decoder is not able to decode a frame from the data available in the input buffer, it will return to the application with the return value ADI_AACD_NEED_MORE_INPUT, asking for more input data.

The decoder implements a finite state machine to keep and track history of where it is up to in the decoding process for a given frame. It uses the return value of the function to notify the application of critical status information (e.g. ADI_AACD_NEED_MORE_INPUT) that needs to be addressed if the decoder is to continue decoding the frame on subsequent calls.

Parameters

Name: `instance_handle`

Type: `adi_aacd_t *`

Direction: Input/Output

Description: Decoder instance.

Name: `num_input_bytes`

Type: `int`

Direction: Input

Description: The number of bytes in the input bitstream for this call to the decoder. The application will consume up to this number of bytes from the input buffer for each call.

Name: `input_bitstream`

Type: `adi_aacd_bitstream_data_t *`

Direction: Input

Description: The AAC coded input bitstream to the decoder. This buffer must contain at least *num_input_bytes*. The decoder will ignore additional data in this buffer.

Name: `dec_output_status`

Type: `adi_aacd_output_status_t *`

Direction: Output

Description: Output status information returned from the decoder. See definition in section 4.3.15 .

Name: `output_pcm_channels[]`

Type: `adi_aacd_audio_data_t *`

Direction: Output

Description: Each element must contain the pointer to where the decoder shall write the output PCM samples of one channel. Each PCM sample is a 16-bit sample and each output channel is required to hold at least **`ADI_AACD_MAX_OUTPUT_PCM_SAMPLES_PER_CHAN`** samples. Additional pointers in the array beyond two will be ignored. PCM data is available in the channel buffers only after the decoder returns ADI_AACD_SUCCESS. Please see Figure 1 on page 25 for more information on interleaving output channels.

Return value: The function returns one of the following error codes:

- ADI_AACD_SUCCESS: The decode operation succeeded and output data is available.
- ADI_AACD_INVALID_POINTER: One of the function arguments was set incorrectly to NULL
- ADI_AACD_NEED_MORE_INPUT: The decode operation requires more data in the input buffer.
- ADI_AACD_INVALID_BITSTREAM: The decoder cannot continue due to bad data.
- ADI_AACD_INVALID_HEADER: The decoder cannot continue due to bad header data.
- ADI_AACD_TERMINAL_FAILURE: The decoder is unable to successfully decode the supplied bitstream.
- ADI_AACD_OPERATION_FAILED: The decoder is unable to successfully decode the supplied payload within the bitstream.
- ADI_AACD_TIMEOUT_REACHED: The decoder library is a demonstration version that has exceeded it's evaluation limit. Output audio will continue with impairments.

4.2.7 adi_aacd_input_is_adts()

Prototype

```
adi_aacd_return_code_t adi_aacd_input_is_adts (
     adi_aacd_bitstream_data_t *input_buf,
     int num_input_bytes
)
```

Description

This function can be called prior to calling adi_aacd_init() to check if the bitstream in the input buffer is encapsulated in the ADTS container format. This function helps to determine if the **`container_format`** field in the configuration data structure is to be set to ADI_AACD_ADTS.

It can also be called prior to calling adi_aacd_resynch () when seeking through an ADTS bitstream, to verify that the start of an ADTS frame has been found before resuming decoding.

Parameters

Name: `input_buf`

Type: `adi_aacd_bitstream_data_t *`

Direction: Input

Description: Pointer to the input buffer to check.

Name: `num_input_bytes`

Type: `int`

Direction: Input

Description: The number of bytes available in the input buffer.

Return value: adi_aacd_return_code_t

- ADI_AACD_SUCCESS = input bitstream is confirmed to be encapsulated in ADTS.
- ADI_AACD_INVALID_POINTER: One of the function arguments was set incorrectly to NULL
- ADI_AACD_NEED_MORE_INPUT = not enough input data was provided to determine if the input bitstream is encapsulated in ADTS.
- ADI_AACD_INVALID_HEADER = input bitstream is NOT encapsulated in ADTS.

4.2.8 adi_aacd_input_is_adif ()

Prototype

```
adi_aacd_return_code_t adi_aacd_input_is_adif (
     adi_aacd_bitstream_data_t *input_buf,
     int num_input_bytes
)
```

Description

This function can be called prior to calling adi_aacd_init() to check if the bitstream in the input buffer is encapsulated in the ADIF container format. This function helps to determine if the **`container_format`** field in the configuration data structure is to be set to ADI_AACD_ADIF.

Parameters

Name: `input_buf`

Type: `adi_aacd_bitstream_data_t *`

Direction: Input

Description: Pointer to the input buffer to check.

Name: `num_input_bytes`

Type: `int`

Direction: Input

Description: The number of bytes available in the input buffer

Return value: adi_aacd_return_code_t

- ADI_AACD_SUCCESS = input bitstream is confirmed to be encapsulated in ADIF.
- ADI_AACD_INVALID_POINTER: One of the function arguments was set incorrectly to NULL
- ADI_AACD_NEED_MORE_INPUT = not enough input data was provided to determine if the input bitstream is encapsulated in ADIF.
- ADI_AACD_INVALID_HEADER = input bitstream is NOT encapsulated in ADIF.

4.3 API Data Structures

4.3.1 adi_aacd_module_info_t

```
typedef struct  {
    unsigned int    version_num;
    char*           version_text;
} adi_aacd_module_info_t;
```

Description

This type is used to store information relating to the module as a whole.

- **`version_num`**

The major-minor-patch version number of this library in the following format:

`0xMMNNPP00`, where 0xMM is the major revision number in hex (MSB), 0xNN is the minor revision number in hex, 0xPP is the patch revision number in hex, and the LSB is reserved to be 0x00.

- **`version_text`**

A pointer to a NULL terminated string which identifies the specific module (and variant), and the version information of this library.

4.3.2 adi_aacd_return_code_t

```
typedef enum {
       ADI_AACD_SUCCESS                        =    0,
       ADI_AACD_TERMINAL_FAILURE               =   -2,
       ADI_AACD_INVALID_BITSTREAM              = -100,
       ADI_AACD_INVALID_HEADER                 = -101,
       ADI_AACD_TIMEOUT_REACHED                =    1,
       ADI_AACD_OPERATION_FAILED               =   10,
       ADI_AACD_INVALID_CONFIG_PARAMS          =   11,
       ADI_AACD_NEED_MORE_INPUT                =  100,
       ADI_AACD_INVALID_POINTER                = 0x10ff
} adi_aacd_return_code_t;
```

Description

ADI_AACD_SUCCESS: The operation was successful.

ADI_AACD_TERMINAL_FAILURE: A critical error occurred and the instance cannot operate until reinitialised.

ADI_AACD_INVALID_BITSTREAM: An unsupported bitstream was encountered.

ADI_AACD_INVALID_HEADER: A corrupted bitstream was encountered.

ADI_AACD_OPERATION_FAILED: The instance failed its last expected operation.

ADI_AACD_INVALID_CONFIG_PARAMS: An incorrect configuration was supplied by the user

ADI_AACD_NEED_MORE_INPUT: The input buffer requires more data before a frame can be processed.

ADI_AACD_TIMEOUT_REACHED: The decoder library is a demonstration version that has exceeded it's evaluation limit. Output will continue at a degraded level, with a 0.5 second beep occurring every 4 seconds in playback.

ADI_AACD_INVALID_POINTER: A NULL pointer was incorrectly supplied by the user to the function that returns this error code.

4.3.3 adi_aacd_bitstream_data_t

```
typedef char adi_aacd_bitstream_data_t;
```

Description

This type is used to represent the input data to the decoder. The size of each element is one byte.

4.3.4 adi_aacd_audio_data_t

```
typedef short adi_aacd_audio_data_t;
```

Description

This type is to be used to represent PCM samples output from the decoder. Each PCM sample produced is a 16-bit integer.

4.3.5 adi_aacd_config_t

```
typedef struct
{
    adi_aacd_audio_config_t               audio_render_params;
    unsigned int                          use_circular_bitstream_buffer;
    int                                   circular_buffer_length;
    adi_aacd_bitstream_data_t *           circular_buffer_base;
    adi_aacd_frame_properties_t *         frame_properties_ptr;
    adi_aacd_stream_properties_t *        stream_properties_ptr;
    adi_aacd_chan_config_t                output_chan_config;
    int                                   output_chan_stride[ADI_AACD_MAX_NUM_OUTPUT_CHANS]
    adi_aacd_stream_format_t              container_format;
    adi_aacd_stream_config_t              container_info;
    adi_aacd_decoder_mode_t               decoding_mode;
    int                                   selected_program_id;
} adi_aacd_config_t;
```

Description

This structure is to be used to store all configuration parameters of the decoder. All parameters shall be assumed to be reconfigurable except where marked as NOT reconfigurable. All non-reconfigurable parameters are only configured once for a given stream, when the adi_aacd_init() function is called.

- **audio_render_params**

Embedded structure parameters specific to this decoder that control the rendering of the audio output. See 4.3.6

- **use_circular_bitstream_buffer**

If nonzero, the input buffer is circular. If zero, the input buffer is linear. (NOT reconfigurable)

- **circular_buffer_length**

The length of the circular buffer (not used by decoder if **use_circular_bitstream_buffer** set to zero). (NOT reconfigurable)

- **circular_buffer_base**

The starting address of the circular buffer (not used by decoder if **use_circular_bitstream_buffer** set to zero). (NOT reconfigurable)

- **frame_properties_ptr**

Pointer to structure holding audio frame-specific information extracted from the bitstream. See 4.3.8 (NOT reconfigurable)

- **stream_properties_ptr**

Pointer to structure holding audio stream-specific information extracted from the bitstream. See 4.3.9 (NOT reconfigurable)

- **output_chan_config**

Allows the application to select the number and configuration of PCM audio channels to be output from the decoder. Note that the HE-AAC v2 stereo decoder only allows

ADI_AACD_2_0_CHAN_CONFIG to be selected when the decoding mode is set to **ADI_AACD_DECODING_MODE_HEAACv2**. See 4.3.10 (NOT reconfigurable)

- **output_chan_stride[]**

Allows the application to set the stride between PCM elements for each of the output PCM channels. Setting this to 1 indicates that the channel buffers are independantly located and not interleaved. Interleaving of channel data could be achieved by using this value to skip over other channel data in the output buffers. (NOT reconfigurable)

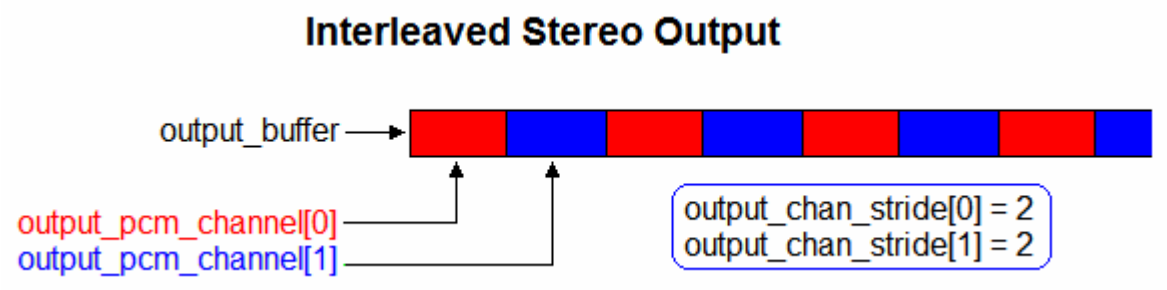

Figure 1 Interleaved Stereo Output

- **container_format**

Allows the application to signal to the decoder which container format encapsulates the input bitstream. See 4.3.11 (NOT reconfigurable)

- **container_info**

Embedded structure containing necessary fields extracted from the bitstream with an external parser. See 4.3.12 (NOT reconfigurable)

- **decoding_mode**:

Allows the decoder to select whether to enable the SBR/PS tools if available in the library variant. For the HE-AAC v2 variant, this switch allows the decoder instance to behave as if it was a stereo AAC-LC decoder, and ignore any SBR information found in the bitstream; or as a stereo HE-AAC v1 (HQ) decoder, and ignore any PS information found in the bitstream. See 4.3.7 (NOT reconfigurable)

- **selected_program_id**

Allows the application to signal to the decoder which audio program to select in the case of multi-program bitstreams. The program configuration element tag ID is used to identify which audio program to decode. The value -1 can also be used for single program bitstreams, or to select the first audio program encountered by the decoder in the bitstream. (NOT reconfigurable)

4.3.6 adi_aacd_audio_config_t

```
typedef struct
{
       unsigned int   enable_dyn_range_control;
} adi_aacd_audio_config_t;
```

Description

This structure (embedded in adi_aacd_config_t) is to control at run-time, how the output audio is rendered.

- **`enable_dyn_range_control`**:

  Flag to allow the user to enable/disable the application of dynamic range compression (default disabled). *In the current version this option has not yet been implemented and the user will not be able to enable this option.*

4.3.7 adi_aacd_decoder_mode_t

```
typedef enum {
    ADI_AACD_DECODING_MODE_AACLC,
    ADI_AACD_DECODING_MODE_HEAACv1,
    ADI_AACD_DECODING_MODE_HEAACv2
} adi_aacd_decoder_mode_t;
```

Description

Enumerated type to select the desired decoding mode. Essentially, this data type controls whether the HE-AAC v2 decoder ignores PS and SBR data in the bitstream.

4.3.8 adi_aacd_frame_properties_t

```
typedef struct
{
       int    sample_rate;
       int    num_channels;
} adi_aacd_frame_properties_t;
```

Description

This structure allows the application to monitor the frame specific information parsed from the bitstream.

- **`sample_rate`**:

  Sample rate of the output audio.

- **`num_channels`**:

  Number of channels encoded in the bitstream.

4.3.9 adi_aacd_stream_properties_t

```
typedef struct
{
       unsigned int   is_sbr_present;
} adi_aacd_stream_properties_t;
```

Description

This structure allows the application to monitor information that applies to all frames parsed from the bitstream.

- **`is_sbr_present`**:

  If nonzero, indicates Spectral Band Replication is present in the stream. This flag is only valid when the configuration parameter decoding_mode is not set to ADI_AACD_DECODING_MODE_AACLC.

4.3.10 adi_aacd_chan_config_t

```
typedef enum {
    ADI_AACD_1_0_CHAN_CONFIG       =    0x1,
    ADI_AACD_2_0_CHAN_CONFIG       =    0x2,
    ADI_AACD_3_0_CHAN_CONFIG       =    0x3,
    ADI_AACD_3_1_CHAN_CONFIG       =    0x5,
    ADI_AACD_3_2_CHAN_CONFIG       =    0x7,
    ADI_AACD_3_2_1_CHAN_CONFIG     =    0x10007
} adi_aacd_chan_config_t ;
```

Description

Enumerated type to select the desired output channel configuration. Must be ADI_AACD_1_0_CHAN_CONFIG (single channel output – data stored by the decoder in the output buffer **`output_pcm_channels[0]`**) or ADI_AACD_2_0_CHAN_CONFIG (two channel output – left channel data stored by the decoder in the output buffer **`output_pcm_channels[0],`** and right channel data in **`output_pcm_channels[1]`**). Using other values in this release will result in an INVALID_CONFIG_PARAMS error from adi_aacd_init().

4.3.11 adi_aacd_stream_format_t

```
typedef enum
{
       ADI_AACD_ADTS,
       ADI_AACD_ADIF,
       ADI_AACD_RAW
} adi_aacd_stream_format_t;
```

Description

This enumeration represents the possible container formats handled by the decoder.

- **ADI_AACD_ADTS:**

  The audio bitstream is encapsulated in the ADTS transport layer. The decoder handles the whole ADTS frame header.

- **ADI_AACD_ADIF:**

  The audio bitstream is encapsulated in the ADIF file format. The decoder handles the whole ADIF file header.

- **ADI_AACD_RAW:**

  The audio bitstream is not encapsulated in any container format. The decoder expects the bitstream to conform to the raw_data_block syntax outlined in the MPEG-4 [1] specification.

4.3.12 adi_aacd_stream_config_t

```
typedef struct
{
    int                num_channels;
    unsigned int       sampling_frequency_index;
    int                sampling_frequency;
    int                sbr_sampling_frequency;
    unsigned int       frame_length_flag;
    unsigned int       channel_configuration;
    unsigned int       num_front_channel_elements;
    unsigned int       num_side_channel_elements;
    unsigned int       num_back_channel_elements;
    unsigned int       num_coupled_channel_elements;
    unsigned int       num_lfe_channel_elements;
    unsigned char      channel_element_id[ADI_AACD_MAX_NUM_AUDIO_ELEMENTS];
} adi_aacd_stream_config_t;
```

Description

This structure is used to allow the application to pass stream configuration information for all container formats other than ADIF/ADTS. The application may use an external parser to extract this information from the file format used to store the audio bitstream, or directly configure the information if in a system that operates on a restricted set of bitstreams conforming to a predefined configuration.

- **num_channels**:

  Total number of audio channels represented in the bitstream (valid values = 0…8). An entry outside this range will cause adi_aacd_init() to return an INVALID_CONFIG_PARAMS error.

- **sampling_frequency_index**:

  A four bit field (as defined in the MPEG4 standard [1]) indicating the sampling rate of the core AAC-LC audio signal used, as shown below.

sampling_frequency_index	Value
0x0	96000
0x1	88200
0x2	64000

sampling_frequency_index	Value
0x3	48000
0x4	44100
0x5	32000
0x6	24000
0x7	22050
0x8	16000
0x9	12000
0xa	11025
0xb	8000
0xc	7350
0xd	reserved
0xe	reserved
0xf	escape value

Table 4-2: Allowed codes for sampling_frequency_index

- **`sampling_frequency`**:

  Configures the actual sampling rate of the core AAC-LC audio signal, if **`sampling_frequency_index`** equals 0xf.

- **`sbr_sampling_frequency`**:

  Configures the actual sampling rate of the output audio signal after applying SBR (must be the same or twice the sampling frequency signalled for the core AAC-LC audio signal).

- **`frame_length_flag`**:

  Flag determining core AAC-LC frame block length: 0 means 1024 samples, 1 means 960 samples.

- **`channel_configuration:`**

  Configures the speaker configuration of the encoded audio channels. This value is signalled in a container format as part of a program configuration element. Valid non-zero values defining a set of default channel configurations are shown below (as defined by ISO-14496-3:2005)

channel_configuration	Meaning
1	center front speaker
2	left, right front speakers
3	center front speaker left, right front speakers
4	center front speaker left, right center front speakers, rear surround
5	center front speaker

channel_configuration	Meaning
	left, right front speakers, left surround, right surround rear speakers
6	center front speaker left, right front speakers, left surround, right surround rear speakers, front low frequency effects speaker
7	center front speaker left, right center front speakers, left, right outside front speakers, left surround, right surround rear speakers, front low frequency effects speaker

Table 4-3: Allowed non-zero codes for channel_configuration

When this value is 0, only the front channel configuration needs to be supplied to the decoder, since the side, back and LFE signals are ignored and not decoded.

- **`num_front_channel_elements:`**

  Configures the number of audio *elements* to be found in the bitstream.carrying front speaker signals (not to be confused with the number of front audio channels). This value is signalled in a container format as part of a program configuration element, and is only used by the decoder when **`channel_configuration`** is set to zero.

- **`num_side_channel_elements:`**

  Configures the number of audio *elements* to be found in the bitstream carrying side speaker signals (not to be confused with the number of side audio channels). This value is signalled in a container format as part of a program configuration element, and is only used by the decoder when **`channel_configuration`** is set to zero. Note any side channel information present in the bitstream is parsed but ignored by the stereo decoder.

- **num_back_channel_elements:**

  Configures the number of audio *elements* to be found in the bitstream carrying rear speaker signals (not to be confused with the number of rear audio channels). This value is signalled in a container format as part of a program configuration element, and is only used by the decoder when **channel_configuration** is set to zero. Note any rear channel information present in the bitstream is parsed but ignored by the stereo decoder.

- **num_coupled_channel_elements:**

  Configures the number of audio *elements* to be found in the bitstream carrying audio signals that are coupled with one or more of the main speaker channels. This value is signalled in a container format as part of a program configuration element, and is only used by the decoder when **channel_configuration** is set to zero. Note any coupled channel information present in the bitstream is parsed but ignored by the stereo decoder.

- **num_lfe_channel_elements:**

  Configures the number of audio *elements* to be found in the bitstream carrying low frequency effects (LFE) speaker signals. This value is signalled in a container format as part of a program configuration element, and is only used by the decoder when **channel_configuration** is set to zero. Note any LFE channel information present in the bitstream is parsed but ignored by the stereo decoder.

- **channel_element_id**:

  An array of bit fields to configure the properties of the audio *elements* in the bitstream.carrying front speaker signals (not to be confused with the number of front audio channels). This information is signalled in a container format as part of a program configuration element, and is only used by the decoder when **channel_configuration** is signalled as zero. Each element of the array is mapped as follows:

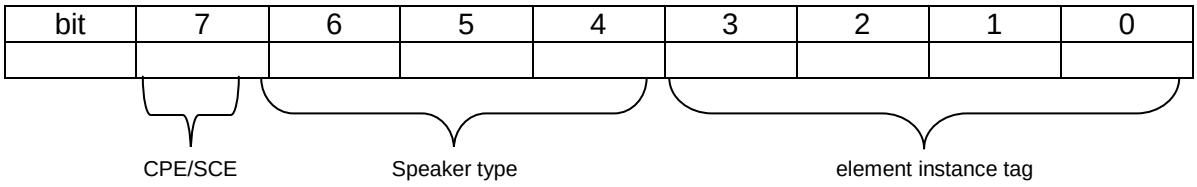

Figure 2: channel_element_id - bit field definitions

bit 7	**CPE/SCE**
0	Identifies an audio element as being a single channel element (contains 1 audio channel)
1	Identifies an audio element as being a channel pair element (contains 2 audio channels)
bits 6:4	**speaker type**
000	reserved
001	Front audio channels
010	Side audio channels

011	Back audio channels
100	LFE audio channel
101 - 111	reserved
bits 3:0	**element instance tag**
xxxx	identifies the audio element having the signalled speaker type and CPE/SCE element type

Table 4-4: `channel_element_id` - bit field definitions

Definitions have been provided in the API header file to help the user set and probe the bit fields of this array. See 4.4.10 for a list of these definitions.

4.3.13 adi_aacd_config_item_t

```
typedef enum {
       ADI_AACD_ALL_RECONFIG_PARAMS         = 0
}
adi_aacd_config_item_t;
```

Description

This enumeration represents the list of configuration parameters that can be reconfigured after the **`adi_aacd_reconfigure`** function has been called. See section 4.2.5 for more details.

4.3.14 adi_aacd_mem_t;

```
typedef struct {
    adi_aacd_state_mem_t      state;
    adi_aacd_scratch_mem_t    scratch;
} adi_aacd_mem_t;
```

Description

This structure contains further structures to describe the state and scratch memory required by the decoder, which need to be passed in via the function adi_aacd_create ().

4.3.15 adi_aacd_output_status_t

```
typedef struct
{
    unsigned int                      warning_info;
    int                               num_output_samples_per_chan;
    unsigned int                      active_output_chans;
    adi_aacd_bitstream_data_t         *unconsumed_input_data_ptr;
    int                               num_input_bytes_consumed;
} adi_aacd_output_status_t;
```

Description

The structure holds information about the last frame of output samples.

- **`warning_info`**:

  Bitmapped warning flags (see 4.4.9 ADI_AACD_WARNING_xxxxx)

- **`num_output_samples_per_chan`**:

  The number of output samples generated for each channel by the decode operation

- **`active_output_chans`**:

  A bit mapped integer that indicates which output channel buffer contains new audio data resulting from the last call to adi_aacd_decoder(). Starting from the least significant bit, bit *n* corresponds to the *n*th buffer configured in the array **`output_pcm_channels`** (see adi_aacd_decoder() and 4.3.10 adi_aacd_chan_config_t).

- **`unconsumed_input_data_ptr`**:

  Pointer to the first unconsumed data byte

- **`num_input_bytes_consumed`**:

  The number of data bytes consumed by the last call to adi_aacd_decoder()

4.3.16 adi_aacd_frame_properties_t

```
typedef struct {
       int    sample_rate;
       int    num_channels;
} adi_aacd_frame_properties_t;
```

Description

- **`sample_rate`**: The sample rate of the output data (samples/second).
- **`num_channels`**: The number of channels of output data

4.3.17 adi_aacd_stream_properties_t

```
typedef struct
{
       unsigned int   is_sbr_present;
} adi_aacd_stream_properties_t;
```

Description

- **`is_sbr_present`**: SBR data was reconstructed by the decoder.

4.4 Macros

4.4.1 ADI_AACD_AACLC_BUILD

One of the following macros will be defined to indicate which variant of the AAC decoder is being integrated into the user system.

- ADI_AACD_AACLC_BUILD
- ADI_AACD_HEAACv2_BUILD

4.4.2 ADI_AACD_MIN_INPUT_NUMBYTES_PER_CHAN

The worst case ratio of the size of the input buffer to the number of channels that is required for correct operation of an ISO 14496-3 conformant decoder.

4.4.3 ADI_AACD_MAX_OUTPUT_PCM_SAMPLES_PER_CHAN

The maximum number of output samples per channel in the output buffer.

4.4.4 ADI_AACD_MAX_NUM_OUTPUT_CHANS

The maximum number of output channels supported (2 for a stereo decoder).

4.4.5 ADI_AACD_MIN_BITSTREAM_BUFFER_NUMBYTES

The minimum size in bytes of the input bitstream buffer required to guarantee the decoder does not run out of data while decoding a frame.

4.4.6 ADI_AACD_ADTS_HEADER_NUMBYTES

The size of an ADTS header as defined in the MPEG4 standard [1].

4.4.7 ADI_AACD_ADIF_HEADER_NUMBYTES

The size of an ADIF header as defined in the MPEG4 standard [1].

4.4.8 ADI_AACD_*xxxx*_*yyyy*_zzzz_MEM_NUMBYTES

The minimum required lengths of the memory type xxxx of priority *yyyy* in memory type *zzzz*.

The memory blocks are allocated to state variables (*zzzz*=STATE, for a memory pool that is unique to each instance) or to scratch variables (*zzzz*=SCRATCH, for a memory pool that can be shared across instances running in the same thread).

Each pool consists of several separate blocks *xxxx* = {fastb0, fastb1, fast, or slow} located in different memory types. Operation is undefined if the memory pools allocated to adi_aacd_create() have blocks smaller than these required lengths.

The variable yyyy is used to aid in the decision of which sections of memory to remove from L1 when space is more important than performance, with a higher number indicating the lowest priority memory objects, and so being the first to be removed from L1 memory.

The defined memory block lengths are:

Name of Macro Definition	AAC Product Variant	
	AAC-LC	HE-AAC v2
ADI_AACD_FASTB0_PRIO0_SCRATCH_MEM_NUMBYTES	(8192)	(16384)
ADI_AACD_FASTB1_PRIO1_SCRATCH_MEM_NUMBYTES	(0)	(0)
ADI_AACD_FAST_PRIO2_SCRATCH_MEM_NUMBYTES	(3248)	(4576)
ADI_AACD_FAST_PRIO3_SCRATCH_MEM_NUMBYTES	(8192)	(24576)
ADI_AACD_FASTB0_PRIO4_STATE_MEM_NUMBYTES	(184)	(7712)
ADI_AACD_FASTB1_PRIO6_STATE_MEM_NUMBYTES	(1872)	(11784)
ADI_AACD_FAST_PRIO7_STATE_MEM_NUMBYTES	(13044)	(13044)

Table 4-5: Size Definitions for the Dynamic Memory Sections

4.4.9 ADI_AACD_WARNING_xxxxx

This set of definitions are bit masks identifying what non-critical error conditions were detected by the decoder. The condition is true when a logical AND of the corresponding mask with the variable **`adi_aacd_output_status_t.warning_info`** is non-zero.

Name of Macro Definition	Meaning
ADI_AACD_WARNING_BAD_FILL_ELEMENT	The audio frame contained a data fill element which did not transmit the expected values. This error condition Is not expected to affect the output audio.

Table 4-6: Warning Flag Definitions

4.4.10 ADI_AACD_ELEMENT_xxxxx

This set of definitions are provided to help the user correctly setup the **channel_element_id** container configuration information.

```
#define ADI_AACD_ELEMENT_IS_FRONT      0x10
#define ADI_AACD_ELEMENT_IS_SIDE       0x20
#define ADI_AACD_ELEMENT_IS_BACK       0x30
#define ADI_AACD_ELEMENT_IS_LFE        0x40
#define ADI_AACD_ELEMENT_IS_CPE        0x80
#define ADI_AACD_ELEMENT_TYPE_MASK     0x70
#define ADI_AACD_ELEMENT_TAG_MASK      0x0F
```

See 4.3.12 for more details.

4.4.11 ADI_AACD_MAX_NUM_AUDIO_ELEMENTS

Maximum number of audio elements carrying front channel signals, handled by the decoder. This definition also represents the size of the **channel_element_id** array. See 4.3.12 for more details.

5 Memory & Performance

5.1 Memory Requirements & Linking

This section describes the memory usage in the AAC decoder library for the Blackfin processor. Some code and data sections should be placed in L1 memory whenever possible to achieve best performance. In the table below, the code and data sections and their ideal placement in memory are outlined. The section names given below also contain a 'prio' value, with 0 being the most important value, which will aid when trying to trade memory footprint for performance. For optimal performance, the instruction and data caches should be enabled.

Section Name	Code or Data	Size (KB)		Ideal Memory Location for Placement
		AAC-LC	HE-AAC v2	
adi_fast_prio0_code	Code	16	20	L1 (On-chip)
adi_fast_prio1_code	Code	0	16	L1 (On-chip)
adi_fast_prio2_code	Code	1	16	L1 (On-chip)
adi_slow_noprio_code	Code	10	32	L2/3 (Off-chip)
<PRIO 0>	Data[Scratch]	see Table 4-5		L1 (On-chip)
<PRIO 1>	Data[Scratch]	see Table 4-5		L1 (On-chip)
<PRIO 2>	Data[Scratch]	see Table 4-5		L1 (On-chip)
<PRIO 3>	Data[Scratch]	see Table 4-5		L1 (On-chip)
<PRIO 4>	Data[State]	see Table 4-5		L1 (On-chip)
adi_fastb0_prio5_r	Data[Static]	0	2	L1 (On-chip)
<PRIO 6>	Data[State]	see Table 4-5		L1 (On-chip)
<PRIO 7>	Data[State]	see Table 4-5		L1 (On-chip)
adi_fastb1_prio8_r	Data[Static]	1	2	L1 (On-chip)
adi_slow_noprio_r	Data[Static]	26	54	L2/3 (Off-chip)

Table 5-1: Module Memory sections

The per instance dynamic memory is required to be supplied by the system to each instance of the decoder via the adi_aacd_create () function. See section 4.4.8 for more details on memory sizes.

5.2 MIPS Performance

Please refer to our Product Highlights document ([8], [9]) for the latest performance figures.

Appendix A: AAC Decoder Usage Example Code

```
Please see the supplied simple file IO example.
```

Code Example 1: Simple usage example for the AAC Decoder