08-6406-API-ZCH66
MAY 20, 2008
2.7

Application Programmers Interface for MPEG-4 AAC LC Decoder

ABSTRACT:

Application Programmers Interface for MPEG-4 AAC LC Decoder

KEYWORDS:

Multimedia codecs, AAC

APPROVED:

Shang Shidong

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	17-Mar-2004	Sanjay	Initial Draft
0.2	18-Mar-2004	Sanjay	Added review comments from Venkat
0.3	03-Apr-2004	Vishala	Updated with review comments from WMSG
1.0			PCS comments incorporated
1.1	10-Sep-2004	Vishala	Internal review comments incorporated
1.2	11-Mar-2005	Vishala	Updated example calling routine
2.0	06-Feb-2006	Lauren Post	Draft version of new format
2.1	09-May-2006	Purusothaman	Added the details of the Debug log function pointer register function.
2.2	12-Dec-2006	Kusuma S	Changed the Callback function prototype
2.3	10-Apr-2008	Baofeng Tian	Add bsac feature to aac
2.4	25-Apr-2008	Bing Song	Add error concealment feature
2.5	20-May-2008	Bing Song	Add version information
2.6	23-June-2008	Haiting Yin	Update for push mode
2.7	15-Oct-2008	Haiting Yin	Remove description for BSAC

Table of Contents

1 Introduction

1.1 Purpose

This document gives the details of the application programmer's interface of MPEG-4 AAC Low Complexity (LC) Decoder. The AAC decoder takes the parsed raw data stream as the input and generates audio PCM samples. The decoder is designed as a set of library routines that read the bit-stream from the input buffers and write the decoded output to the output buffers. The AAC decoder implementation currently supports LC (Low Complexity) profile with single channel ('mono') and two-channels ('stereo') decoding.

1.2 Scope

This document describes only the functional interface of the AAC decoder. It does not describe the internal design of the decoder. Specifically, it describes only those functions by which a software module can use the decoder.

1.3 Audience Description

The intended audience for this document is the development community who wish to use the AAC decoder in their systems. The reader is expected to have basic understanding of Audio Signal processing and AAC decoding.

1.4 References

1.4.1 Standards

- ISO/IEC 13818-7:1997 Information technology -- Generic coding of moving pictures and associated audio information -- Part 7 (popularly known as *MPEG-2 AAC*)
- ISO/IEC 13818-4:1997 Information technology -- Generic coding of moving pictures and associated audio information -- Part 4 (compliance testing)
- ISO/IEC 14496-3:1999 Information technology – Coding of audio visual objects -- Part 3 (audio)
- ISO/IEC 14496-14:2003, Information technology -- Coding of audio-visual objects -- Part 14:MP4 file format
- ISO/IEC 14496-12:2005, Information technology -- Coding of audio-visual objects -- Part 12: ISO base media file format

1.4.2 General References

- Ted Painter and Andreas Spanias, "Perceptual Coding of Digital Audio", Proc. IEEE, vol-88, no.4, April 2000

- H.S.Malvar, “Lapped transforms for efficient subband/transform coding”, IEEE trans. ASSP, June 1990.
- Seymour Shlien, “The Modulated Lapped Transform, Its Time-Varying Forms and Its Applications to Audio Coding Standards.”
- “A Tutorial on MPEG/Audio compression” by Davis Pan

1.4.3 Freescale Multimedia References

- AAC Decoder Application Programming Interface – aac_dec_api.doc
- AAC Decoder Requirements Book - aac_dec_reqb.doc
- AAC Decoder Test Plan - aac_dec_test_plan.doc
- AAC Decoder Release notes - aac_dec_release_notes.doc
- AAC Decoder Test Results – aac_dec_test_results.doc
- AAC Decoder Performance Results – aac_dec_perf_results.doc
- AAC Decoder Interface header – aacd_dec_interface.h
- AAC Decoder Application Code – aac_main.c

1.5 Definitions, Acronyms, and Abbreviations

TERM/ACRONYM	DEFINITION
AAC	Advanced Audio Coding
ADIF	Audio_Data_Interchange_Format
ADTS	Audio_Data_Transport_Stream
API	Application Programming Interface
ARM	Advanced RISC Machine
FSL	Freescale
IEC	International Electro-technical Commission
ISO	International Standards Organization
LC	Low Complexity
MDCT	Modified Discrete Cosine Transform
MPEG	Moving Pictures Expert Group
OS	Operating System
PCM	Pulse Code Modulation
PNS	Perceptual Noise Substitution
UNIX	Linux PC x/86 C-reference binaries
RVDS	ARM RealView Development Suite
TBD	To be decided

1.6 Document Location

docs/aac_dec

2 API Description

This section describes the steps followed by the application to call the AAC decoder. During each step the data structures used and the functions used will be explained. Pseudo code is given at the end of each step. The member variables inside the structures are prefixed as aacd_ or app_ to indicate if that member variable needs to be initialized by the decoder or application.

Step 1: Allocate memory for Decoder parameter structure

The application allocates memory for the structure mentioned below. This structure contains the decoder parameters and memory information structures.

```
/* Decoder  parameter  structure */
typedef struct {
    AACD_Mem_Alloc_Info    aacd_mem_info;
    void *aacd_decode_info_struct_ptr;
    AACD_UINT8   num_pcm_bits;
    long         *AACD_bno;
    AACD_Block_Params *params;
    AACD_Ch_Info ch_info[Chans];
    AACD_INT32 adts_format;
    AACD_INT32 packet_loss;
} AACD_Decoder_Config;
```

Description of the decoder parameter structure *AACD_Decoder_Config*

aacd_mem_info

This is a memory information structure. The application needs to call the function aacd_query_dec_mem to get the memory requirements from the decoder. The decoder will fill this structure with its memory requirements. This will be discussed in **Step 2**.

aacd_decode_info_struct_ptr

This is a void pointer. This will be initialized by the decoder during the initialization routine. This will then be a pointer to a structure which contains the pointers to tables, buffers and symbols used by the decoder.

num_pcm_bits

The application has to indicate the decoder the output precision required to be outputted. The decoder can output the PCM samples either as 16bit samples or as 24 bit samples.

`AACD_bno`

Point to frame counter.

params

This is a pointer to a structure containing the header information. The header is parsed by the application and this structure is filled with the appropriate header information. This structure needs to be updated whenever header is parsed. This will be discussed in **step 4**

`ch_info`

Channel information got from header.

`adts_format`

Used for error concealment to indicate if the stream is adts format.

`packet_loss`

Used for error concealment. If the flag set as AACD_PACKET_LOSS, the decoder will output one frame data based on previous frame data.

Example pseudo code for this step

```
/* Allocate memory for the local array of table pointers */
   void *aptable[58];

/* Allocate memory for the decoder parameter */
   dec_config = (AACD_Decoder_Config *)
                                alloc (sizeof(AACD_Decoder_Config);

/* Initialize aptable with the start address of all the tables used
   for AAC Decoder. The start addresses of all the tables are available
   in the header file aacd_tables.h  */

/* Fill up the Relocated data position */
   dec_config->app_initialized_data_start = (void *) aptable;

/* Request the output PCM samples to be in the 16bit format/24bit format
   dec_config->num_pcm_bits = AACD_24_BIT_OUTPUT;

/* Assign swap-buffer function to the swap buffer function pointer */
   dec_config->app_swap_buf = app_swap_buffers_aac_dec;

/* Get stream type */
if (App_adts_header_present)
{
    dec_config->adts_format = AACD_TRUE;
}
else
{
    dec_config->adts_format = AACD_FALSE;
}
```

Step 2: Get the decoder version information

This function returns the codec library version information details. It can be called at any time and it provides the library's information: Component name, supported ARM family, Version Number, supported OS, build date and time and so on.

The function prototype of *aacd_decode_versionInfo* is:

C prototype:

```
const AACD_INT8 * aacd_decode_versionInfo();
```

Arguments:

- None.

Return value:

- const char * - The pointer to the constant char string of the version information string.

Example pseudo code for the memory information request

```
{
  // Output the AAC Decoder Version Info
  printf("%s \n", aacd_decode_versionInfo());
}
```

Step 3: Get the decoder memory requirements

The AAC decoder does not do any dynamic memory allocation. The application calls the function *aacd_query_dec_mem* to get the decoder memory requirements. This function must be called before all other decoder functions are invoked.

The function prototype of *aacd_query_dec_mem* is :

C prototype:

```
AACD_RET_TYPE aacd_query_dec_mem (AACD_Decoder_config * dec_config);
```

Arguments:

- dec_config - Decoder config. pointer.

Return value:

- AACD_OK - Memory query successful.
- Other codes - Error (For other error codes refer to appendix).

This function populates the memory information structure, which is described below:

Memory information structure array

```
typedef struct {
      /*  Number of valid memory requests */
      AACD_INT32                  aacd_num_reqs;
      AACD_Mem_Alloc_Info_Sub     mem_info_sub[MAX_NUM_MEM_REQS];
} AACD_Mem_Alloc_Info;
```

Description of the structure *AACD_Mem_Alloc_Info*

aacd_num_reqs

The number of memory chunks requested by the decoder.

mem_info_sub

This structure contains each chunks' memory configuration parameters.

```
typedef struct {
      AACD_INT32       aacd_size; /* Size in bytes */
      AACD_INT32       aacd_type; /* Memory type Fast or Slow */
      AACD_MEM_DESC    aacd_mem_desc;
            /* Flag to indicate Static/Scratch memory */
      AACD_MEM_PRIORITY   aacd_priority;
```



```
            /* In case of fast Memory type, specify the priority */
      void  *app_base_ptr;
      /* Pointer to the base memory, which will be allocated
            And filled by the application*/
} AACD_Mem_Alloc_Info_sub
```

Description of the structure *AACD_Mem_Alloc_Info_sub*

aacd_size

The size of each chunk in bytes.

aacd_type

The type of the memory indicates if the requested chunk of memory needs to be allocated in external or internal memory. The type of memory can be SLOW_MEMORY or external memory, FAST_MEMORY or internal memory. In targets where there is no internal memory, the application can allocate memory in external memory. (Note: If the decoder requests for a FAST_MEMORY for which the application allocates a SLOW_MEMORY, the decoder will still decode, but the performance (MHz) will suffer.)

aacd_mem_des

The memory description field indicates whether requested chunk of memory is static or scratch.

aacd_priority

In case, if the decoder requests for multiple memory chunks in the fast memory, the priority indicates the order in which the application has to prioritize placing the requested chunks in Fast memory

app_base_ptr

This will be initialized by the application. The application will allocate the memory for each chunk depending on the requested size and the type, and then assign the base address of this chunk of memory to *app_base_ptr*. The application should allocate the memory that is aligned to a 4 byte boundary in any case.

Example pseudo code for the memory information request

```
/* Query for memory */
retval = aacd_query_dec_mem (&dec_config);

if (retval != AACD_OK)
      return 1;
```

Step 4: Allocate Data Memory for the decoder

In this step the application allocates the memory as required by the AAC Decoder and fills up the base memory pointer *'app_base_ptr' of 'AACD_Mem_Alloc_Info_sub*' structure for each chunk of memory requested by the decoder.

Example pseudo code for the memory allocation and filling the base memory pointer by the application.

```
AACD_Mem_Alloc_Info_sub *mem;

/* Number of memory chunks requested by the decoder */
nr = dec_config->aacd_mem_info.aacd_num_reqs;
```

```
for(i = 0; i < nr; i++)
{
	mem = &(dec_config-> aacd_mem_info.mem_info_sub[i]);
	if (mem->aacd_type == FAST_MEMORY)
	{
		/* This function allocates memory in internal memory */
		mem->app_base_ptr = alloc_fast(mem->aacd_size);
	}
	else
	{
		This function allocates memory in external memory */
		mem->app_base_ptr = alloc_slow(mem->aacd_size);
	}
}
```

The functions alloc_fast and alloc_slow are required to allocate the memory aligned to 4 byte boundary.

Step 5: Get the header information

The application does the parsing of the header. The header structure which is required by the decoder to process the raw data is updated by the application. Please refer to [**Error! Reference source not found.**] for more details about the structures given below.

Header information structure:


```
typedef struct
{
      AACD_INT32        num_pce;
      AACD_ProgConfig *pce;

      AACD_INT32 ChannelConfig;
      AACD_INT32 SamplingFreqIndex;
      AACD_INT32 BitstreamType;
      AACD_INT32 BitRate;
      AACD_INT32 BufferFullness;
            /*No. of bits in decoder buffer after decoding
            First raw_data_block */
      AACD_INT32 ProtectionAbsent;
      AACD_INT32 CrcCheck;
      AACD_INT32 frame_length;

#ifdef OLD_FORMAT_ADTS_HEADER
      AACD_INT32 Flush_LEN_EMPHASIS_Bits;
#endif
      AACD_INT32 scalOutObjectType;
      AACD_INT32 scalOutNumChannels;
      AACD_INT32 sampleRate;
      AACD_INT32 framelengthflag;
      AACD_INT32 iMulti_Channel_Support;
      AACD_INT32 bsacDecLayer;
} AACD_Block_Params;
```

Description of the structure *AACD_Block_Params*

num_pce

Number of program config elements in the header. If there are no program config elements in the header, num_pce should be set to zero.

pce

Pointer to the Program config structure. (Details of Program config structure is given below).

ChannelConfig

Channel Configuration parameter can take values in the range 0-7. Each value specified indicates preset channel-to-speaker mapping defined in the standard. If num_pce is greater than zero, then ChannelConfig will be ignored as pce will be used.

SamplingFreqIndex

This is an index into an array of valid sampling-frequency values. This will be Ignored if num_pce is greater than zero.

BitstreamType

0 = Constant bit rate (CBR)
1 = Variable bit rate (VBR)

BitRate

If BitstreamType = CBR, then it indicates actual bit rate
If BitstreamType = VBR, then it indicates peak bit rate if not equal to zero
If BitstreamType = VBR and BitRate = 0, then peak bit rate is unknown.

BufferFullness

No. of bits remaining in the encoder buffer after encoding the first raw_data_block in the frame. This will be ignored if num_pce is greater than zero.

ProtectionAbsent

If ProtectionAbsent =1, then CrcCheck value will not be present in the stream

If ProtectionAbsent =0, then CrcCheck value will be present in the stream and CrcCheck will be done by the decoder.

CrcCheck

32bit field used to perform CRC check by the decoder.

frame_length

Bytes in one frame which get from ADTS header.

Flush_LEN_EMPHASIS_Bits

This is used only for the MPEG4 streams with old format ADTS headers

0 : If byte alignment is already existing at the end of the header

1: If there is no byte alignment at the end of the header.

If this is 1, decoder flushes 2 bits before decoding every frame.

. Below elements is used for bsac decoding:

scalOutObjectType

This is used for identify bsac type bitstream, for bsac type, the value should be 22.

scalOutNumChannels

This is used for bsac output channels, it should be less or equal than 2.

sampleRate

This is used to indicate current bitstream's sample rate.

Framelengthflag

This is used to indicate the frame size

0: current frame size is 1024.

1: current frame size is 960.

iMulti_Channel_Support

This is used to indicate whether the decoder need to decode channel than two. This is for future use, currently, we do not support this feature.

bsacDecLayer

This is used to indicate how many enhanced layer you want to decode.currently bsac decoder are not supported in this release.

-1: decode all layers that in current bitstream.

0: only decode base layer

Others: decode specific layers of enhanced layers

```
PCE Structure:
typedef struct
{
    AACD_INT32        profile;
    AACD_INT32        sampling_rate_idx;
    AACD_EleList      front;
    AACD_EleList      side;
    AACD_EleList      back;
    AACD_EleList      lfe;
    AACD_EleList      data;
    AACD_EleList      coupling;
    AACD_MIXdown      mono_mix;
    AACD_MIXdown      stereo_mix;
    AACD_MIXdown      matrix_mix;
    AACD_INT8         comments[1];
    AACD_INT32        buffer_fullness;
    AACD_INT32        tag;
}AACD_ProgConfig;
```


Description of the structure *AACD_ProgConfig*

profile
This should be set to 1 always (MPEG4 AAC Low Complexity profile)

sampling_rate_idx
This is an index into an array of valid sampling-frequency values

front
Structure containing information about front channel elements.

side
Structure containing information about side channel elements.

back
Structure containing information about back channel elements.

lfe
Structure containing information about low frequency enhancement channel elements.

data
Structure containing information about the associated data elements for this program

coupling
Structure containing information about the coupling channel elements.

mono_mix
Structure containing information about the mono_mixdown element

stereo_mix
Structure containing information about the stereo mixdown element

matrix_mix
Structure containing information about the matrix mixdown element.

comments
General information

buffer_fullness
No. of bits remaining in the encoder buffer after encoding the first raw_data_block in the frame.

tag
ID of the PCE

EleList structure:

```
typedef struct
{
    AACD_INT32        num_ele;
    AACD_INT32        ele_is_cpe[16];
    AACD_INT32        ele_tag[16];
}  AACD_EleList;
```

Description of the structure *AACD_EleList*

num_ele
Number of elements in the channel

ele_is_cpe
Indicates whether the element is channel_pair or not

ele_tag
Unique id for the channel element.

Mixdown structure:


```
typedef struct
{
    AACD_INT32          present;
    AACD_INT32          ele_tag;
    AACD_INT32          pseudo_enab;
}  AACD_MIXdown;
```

Description of the structure *AACD_MIXdown*

present

Indicates if the mix_down element is present or not

ele_tag

Indicates the number of the mix_down element

pseudo_enab

This is defined only for matrix mix-down and indicates if pseudo-surround is enabled or not.

Step 6: Memory allocation for input buffer

AAC decoder use push mode when reading the input data, the input buffer should be large enough so that the decoder can generate at least 1 frame of output. For adts file the least input buffer size can be extracted from the adts frame header, and for adif file, the least input buffer size should be larger than 768*Ch, while Ch is the channel number.

Step 7: Initialization routine

All initializations required for the decoder are done in *aacd_decode_init*. This function must be called before the main decoder function is called.

C prototype:
*AACD_RET_TYPE aacd_decode_init (AACD_Decoder_Config *);*

Arguments:

- Decoder parameter structure pointer.

Return value:

- AACD_OK - Initialization successful.
- Other codes - Initialization Error

Example pseudo code for calling the initialization routine of the decoder

```
/* Initialize the AAC decoder. */
retval = aacd_decode_init (dec_config);

if (retval != AACD_OK)
      return 1;
```


Step 8: Memory allocation for output buffer

The application has to allocate memory for the output buffers to hold the decoded stereo PCM samples for a maximum of one frame size. The pointer to this output buffer needs to be passed to the aacd_decode_frame function. The application can allocate memory for output buffer in external memory using alloc_slow. Allocating memory in internal memory using alloc_fast will improve the performance (MHz) of the decoder marginally. It would be desirable to allocate the buffer in the slow memory.

Example pseudo code for allocating memory for output buffer

```
/* Allocate memory for output buffer */
      outbuf = alloc_slow(<number_of_channels>*AACD_FRAME_SIZE);
```

In the example code, the output buffer has been declared as a two dimensional array

```
      AACD_OutputFmtType Outbuf[num_of_channels][AACD_FRAME_SIZE];
```

Note: The current implementation of MPEG4 AAC Decoder supports only LC profile. Hence the maximum number of channels is two. If the output is mono, the output will be generated in outbuf[0][0]. If the output is stereo, the two channels output will be generated in outbuf[1][0] and outbuf[2][0].

Step 9: Call the frame decode routine

The main AAC decoder function is *aacd_decode_frame*. This function decodes the AAC bit stream in the input buffer to generate one frame of decoder output per channel in each call. Be sure that the input buffer should contain enough data for the decoder to generate one frame of output samples.

The output buffer is first filled with left channel samples and then with the right channel, samples. For mono streams, the decoder fills only the left channel samples and leaves the right channel samples unfilled.

The decoder fills up the structure AACD_Decoder_Info.

```
typedef struct {
      AACD_UINT32 aacd_sampling_frequency;
            /* Sampling frequency of the current frame in KHz  */
      AACD_UINT32 aacd_num_channels;
            /* Number of channels decoded in current frame */
      AACD_UINT32 aacd_frame_size;
            /* Number of stereo samples being output for this frame  */
      AACD_UINT32 aacd_len;                        /*    */
      AACD_UINT32       aacd_bit_rate;               /*     */
      AACD_UINT32       BitsInBlock;
            /* Decoder outputs the number of bits in the raw data block
            decoded. This can be used by the application to keep track of
            the bitstream */
} AACD_Decoder_Info;
```

If the bit stream has errors, the decoder will return the corresponding error (mentioned in the appendix), except for the AACD_END_OF_STREAM.

C prototype:

```
AACD_RET_TYPE  aacd_decode_frame (  AACD_Decoder_Config *dec_config,
                                    AACD_Decoder_Info *dec_info,
                                    AACD_OutputFmtType *output_buffer
                                    AACD_INT8 * inbuf, AACD_INT32 len);
```

Arguments:

- dec_config Decoder parameter structure pointer
- dec_info Decoder output parameter pointer
- output_buffer Pointer to the output buffer to hold the decoded samples
- inbuf Pointer to the input buffer containing the raw data block
- len Length of the input buffer

Return value:

- AACD_OK indicates decoding was successful.
- **Others indicates error**

When the decoder encounters the end of bit-stream, the application comes out of the loop. In case of error while decoding the current frame, the application can just ignore the frame without processing the output samples by continuing the loop.
In case of mono bit-streams, as mentioned earlier, the decoder fills only the left channel. It is the responsibility of the application to use it accordingly. One example is the case in which the application can copy the left channel samples into right channel. This is illustrated in the example code below.

Example pseudo code for calling the main decode routine of the decoder

```
AACD_Decoder_Info   dec_info;
AACD_INT8 inbuf;
AACD_UINT32 len;

while (TRUE)
{
      /* Decode one frame */
      /* The decoded parameters for this frame are available in the
         structure AACD_Decoder_info */
      /* application determine below length */
      len = input buffer's length;

      /* Get the input raw data block */

      /* Read(inbuf, len); */

       if (packer_loss)
      {
            dec_config->packet_loss = AACD_PACKET_LOSS;
      }

      retval = aacd_decode_frame (dec_config, &dec_info,
                                    outbuf, inbuf, len);
```



```
        if (retval == AACD_END_OF_STREAM)
        {
              /* Reached the end of bit-stream */
              break;
        }

        if (retval != AACD_OK)
        {
              /* Invalid frame encountered, do not output */
             continue;
        }

        /* If mono, copy the left channel to the right channel. */
        if (dec_info.aacd_num_channels == 1)
        {
              int i;
              for (i = 0; i < dec_info.aacd_frame_size; i++)
                  outbuf [AACD_FRAME_SIZE+i] = outbuf [i];
        }

        /* The output frame is ready for use. Decoding of the next frame
        should start only if the previous output frame has been fully used
        by the application. */

        /* audio_output_frame () is an application function that outputs
        the decoded samples to the output port/device. This function is not
        given in this document. */

        audio_output_frame(outbuf, 2*AACD_FRAME_SIZE);
}
```

Step 10: Free memory

The application releases the memory that it allocated to AAC Decoder if it no longer needs the decoder instance.

```
free (outbuf);
free (inbuf);
for (i=0; i<nr; i++)
{
      free (dec_config-> aacd_mem_info.mem_info_sub[i].app_base_ptr);
}
free (dec_config);
```

3 Example calling Routine

Example for aac lc:

```
#include < aacd_dec_interface.h>
int main(int argc, char *argv[])
{
    AACD_RET_TYPE rc;
    int Length, temp_len;

    AACD_Decoder_Config *dec_config;
    AACD_Decoder_info dec_info;
    AACD_Mem_Alloc_Info_Sub   *mem;
    int                       nr;
    int                       rec_no;
    char opath[AACD_PATH_LEN];
    char InFileName[AACD_PATH_LEN];
    char TestVectors[][AACD_PATH_LEN] = {"L5_48000.aac",
"_end_of_vectors_"};

    int ShouldOpen; /* For managing file opening by AACD_writeout
*/
    int ShouldClose; /* For managing file opening by AACD_writeout
*/

    int  n_bytes;

    {
     // Output the AAC Decoder Version Info
     printf("%s \n", aacd_decode_versionInfo());
    }
    //Check Command-line arguments.
    if (argc > 1) /* command-line arguments were given */
    {
        if (argc > 4)
        {
            printf("Usage : %s  (This runs only the first test-
vector defined in TestVectors[]) \n", argv[0]);
            printf("      : %s <infile> <outfile> <output bits per
sample>\n", argv[0]);
            exit(1);
        }
    }

    if (argc < 3)
    {
        printf ("Usage: %s <infile> <outfile> <output bits per
sample>\n", argv[0]);
        exit (1);
    }
//Allocate memory for the config structure
    dec_config = (AACD_Decoder_Config *) aacd_alloc_fast
(sizeof(AACD_Decoder_Config));
```

```
    if (dec_config == NULL)
        return 1;

//Query memory
    if( aacd_query_dec_mem (dec_config) != AACD_ERROR_NO_ERROR)
    {
        printf("Failed to get the memory configuration for the
decoder\n");
        return 1;
    }
    /* Number of memory chunk requests by the decoder */
    nr = dec_config->aacd_mem_info.aacd_num_reqs;

    for(rec_no = 0; rec_no < nr; rec_no++)
    {
        mem = &(dec_config->aacd_mem_info.mem_info_sub[rec_no]);

        if (mem->aacd_type == AACD_FAST_MEMORY)
        {
            mem->app_base_ptr = aacd_alloc_fast (mem->aacd_size);

            if (mem->app_base_ptr == NULL)
                return 1;
        }
        else
        {
            mem->app_base_ptr = aacd_alloc_slow (mem->aacd_size);

            if (mem->app_base_ptr == NULL)
                return 1;
        }
           memset(dec_config-
>aacd_mem_info.mem_info_sub[rec_no].app_base_ptr,
                0, dec_config-
>aacd_mem_info.mem_info_sub[rec_no].aacd_size); //TLSbo74884

    }


            ShouldOpen = 1;
            ShouldClose = 0;


            bitstream_buf_index = 0;
            bitstream_count = 0;
            bytes_supplied = 0;
            BitsInHeader=0;
            App_adif_header_present = 0;
            App_adts_header_present = 0;

            dec_config->num_pcm_bits     = AACD_16_BIT_OUTPUT;
           if (argc == 4)
           {
                    if(strcmp(argv[3],"16")==0)
                {
```

```
                        dec_config->num_pcm_bits     =
AACD_16_BIT_OUTPUT;
                    }
                        else if(strcmp(argv[3],"24")==0)
                        {
                        dec_config->num_pcm_bits     =
AACD_24_BIT_OUTPUT;
                        }
                        else
                        {
                        printf ("Usage: %s <infile> <outfile>
<output bits per sample>\n", argv[0]);
                        printf ("<output bits per sample> is 16 or
24\n");
                        exit (1);
                        }
              }

               if ((argc == 1)||(argc == 0)) /* No command line
arguments were given */
               {
                   strcat(InFileName,TestVectors[0]);
                   strcat(opath, TestVectors[0]);
              strcat(opath, "_f00.hex");
               }
               else
               {
                   strcat(InFileName, argv[1]);
                   strcat(opath, argv[2]);
               }

               printf(" Initializing decoder..");
               rc = aacd_decoder_init(dec_config);


               printf("Done\n");

               fin = fopen (InFileName, "rb");
               if (fin == NULL)
               {
                   printf ("Couldn't open input file %s\n",
InFileName);
                   exit (1);
               }


               in_buf_done = 0;

               if (strlen(opath) + strlen("_f00.pcm") + 1 >
AACD_PATH_LEN)
               {
                  myexit(AAC_ERROR_FILEIO);  //DSPhl28187
               }

               strcpy(dec_info.output_path,opath);
               fseek(fin, 0, SEEK_END);
```

```
            /*get the stream to the buffer*/
            bitstream_count = ftell(fin);

            if (bitstream_count > MAX_ENC_BUF_SIZE)
            {
                printf("Application Error: InputFileSize >
InputBufferSize\n");
                printf("Application Error: \nInputFileSize=%d
\nInputBufferSize=%d\n", bitstream_count, MAX_ENC_BUF_SIZE);
                exit(1);
            }

            //rewind(fin);
                 fseek(fin, 0, SEEK_SET);
            n_bytes = fread(bitstream_buf, 1, bitstream_count,
fin);
            if( n_bytes != (int)bitstream_count )
//tlsbo89610
            {
                printf("Bitstream not read correctly\n");
                return 1;
            }


            /******************  Decoding Begins
**********************/

            Length = prepare_bitstream();
            App_bs_readinit(bitstream_buf+(bitstream_buf_index-
Length), Length);

            FileType = App_bs_look_bits(32);

            if (App_FindFileType(FileType) != 0)
            {
                printf("InputFile is not AAC\n");
                exit(1);
            }

            update_bitstream_status(0);

            if (App_adif_header_present)
            {
                Length = prepare_bitstream();
                App_bs_readinit(bitstream_buf+bitstream_buf_index-
Length, Length);
                BitsInHeader = 0;
                App_get_adif_header(&BlockParams);
                dec_config->params = &BlockParams;
                update_bitstream_status(BitsInHeader/8);
                 memcpy (input_buff,
bitstream_buf+bitstream_buf_index-Length, Length);
                 bitstream_count_copy_once =0;
                pinput_buff = input_buff;
                      temp_len = Length;
                      dec_info.BitsInBlock = 0;
```

```
                    dec_config->adts_format = AACD_FALSE;
          }


          /* Get ADTS-Header if present and start decoding */

          for(;;)
          {
              if (bitstream_count <= 0)
              {
                  ShouldClose = 1;
                  //DSPhl28187
                         break;
             {
             int i,j;
          if (AACD_24_BIT_OUTPUT == dec_config->num_pcm_bits)
          {

             for(i=0;i<dec_info.aacd_len;i++)
             {

                   for(j=0;j<CHANS;j++)
                   {
                   if(*(dec_config->ch_info[j].present))
                         {
                         fwrite(&outbuf[j][i], 3, 1, pfOutput);
                         if(1 == dec_info.aacd_num_channels)
                               fwrite(&outbuf[j][i], 3, 1, pfOutput);

                         }

                   }
             }
      }
             else
      {
             for(i=0;i<dec_info.aacd_len;i++)
             {

                   for(j=0;j<CHANS;j++)
                   {
                               if(*(dec_config->ch_info[j].present))
                         {
                               fwrite(&outbuf[j][i], 2, 1, pfOutput);
                               if(1 == dec_info.aacd_num_channels)
                               fwrite(&outbuf[j][i], 2, 1, pfOutput);
                         }
                   }
             }
      }
             }
                   }

                   if (App_adts_header_present)
                   {
                       BitsInHeader = 0;
```

```
                    Length = prepare_bitstream();

App_bs_readinit(bitstream_buf+bitstream_buf_index-Length, Length);
                    App_get_adts_header(&BlockParams);
                    dec_config->params = &BlockParams;
                    update_bitstream_status(BitsInHeader/8);
                    dec_config->adts_format = AACD_TRUE;
                }

                 if (App_adts_header_present)
                      {
                      Length =
prepare_bitstream_adts(&BlockParams);

                memcpy (input_buff,
bitstream_buf+bitstream_buf_index-Length, Length);
                    pinput_buff = input_buff;
                    temp_len = Length;

                      }
                       else if
((App_adif_header_present)&&(*(dec_config->AACD_bno) > 0))
                           {
                           int bytes_left;
                           Length = prepare_bitstream_adif();
                           bytes_left = Length -
bitstream_count_copy_once;

                           if (bytes_left < AACD_6CH_FRAME_MAXLEN)
                                {
                               memcpy (input_buff,
bitstream_buf+bitstream_buf_index-Length, Length);
                        bitstream_count_copy_once = 0;
                        pinput_buff = input_buff;
                                 temp_len = Length;
                                 }
                    else
                      {
                      pinput_buff += dec_info.BitsInBlock/8;
                                 temp_len     -=
dec_info.BitsInBlock/8;
                      }

                           }


                rc = aacd_decode_frame(dec_config,&dec_info,outbuf,
pinput_buff, temp_len);


                if (rc != AACD_ERROR_NO_ERROR)
                {
                    App_display_error_message(rc);
                    ShouldClose = 1;
```

```
                                if (rc == AACD_ERROR_EOF)

                                      break;
                                if (App_adif_header_present)
                                      break;

                        }

                        update_bitstream_status(dec_info.BitsInBlock/8);

                        if (App_adif_header_present)
                        {
                                bitstream_count_copy_once +=
dec_info.BitsInBlock/8;
                        }

                        if (*(dec_config->AACD_bno) > 1)
                        {
                            //ptr->out_ptr = opath; //TLSbo63933
                            //DSPhl28187
                        }

                        if (rc == AACD_ERROR_EOF)
                        {
                            App_display_error_message(rc);
                            break;
                        }
                        if (*(dec_config->AACD_bno) > 1)
                        {
                                                if (*(dec_config->AACD_bno)
== 4)                                {
                                // To get the parameters in Log file



                            }

                            fflush (stdout);
                        }
                      //  printf ("Frame No = [%3d]\n", (int) ptr-
>AACD_bno);
                        fflush (stdout);
                    } /*end-while*/



                    fclose(fin);


            for(rec_no = 0; rec_no < nr; rec_no++)
            {
                mem = &(dec_config->aacd_mem_info.mem_info_sub[rec_no]);

                if (mem->app_base_ptr)
                {
                    aacd_free(mem->app_base_ptr);
```

```
            mem->app_base_ptr = 0;
        }
    }
    aacd_free(dec_config);
    return 0;
}
/*****************************************************************
**********
 *
 *   FUNCTION NAME - prepare_bitstream
 *
 *   DESCRIPTION
 *       This function, sets bitstream variables, so that any
subsequent call
 *       to either app_swap_buffer_aac_dec() or
update_bitstream_status()
 *       will work correctly.
 *
 *   ARGUMENTS
 *       None.
 *
 *   RETURN VALUE
 *       Number of bytes available in the buffer, for any
subsequent call
 *       to any stream-parsing routine
 *

*****************************************************************
*********/
int prepare_bitstream()
{
  int len;

  len = (bitstream_count > BS_BUF_SIZE) ? BS_BUF_SIZE :
bitstream_count;
  bitstream_buf_index += len;
  bitstream_count     -= len;
  in_buf_done         += len;
  bytes_supplied      += len;

  return(len);
}


/*****************************************************************
**********
 *
 *   FUNCTION NAME - prepare_bitstream_adts
 *
 *   DESCRIPTION
 *       This function, sets bitstream variables, so that any
subsequent call
 *       to either app_swap_buffer_aac_dec() or
update_bitstream_status()
 *       will work correctly.
 *
```

```
 *   ARGUMENTS
 *       None.
 *
 *   RETURN VALUE
 *       Number of bytes available in the buffer, for any
subsequent call
 *       to any stream-parsing routine
 *

*****************************************************************
*********/
int prepare_bitstream_adts(AACD_Block_Params * params)
{
  int len, frame_len;
  len = 0;
  frame_len = params->frame_length;
  len = (bitstream_count > frame_len) ? frame_len :
bitstream_count;
  bitstream_buf_index += len;
  bitstream_count     -= len;
  in_buf_done         += len;
  bytes_supplied      += len;

  return(len);
}
/*****************************************************************
**********
 *
 *   FUNCTION NAME - prepare_bitstream_adts
 *
 *   DESCRIPTION
 *       This function, sets bitstream variables, so that any
subsequent call
 *       to either app_swap_buffer_aac_dec() or
update_bitstream_status()
 *       will work correctly.
 *
 *   ARGUMENTS
 *       None.
 *
 *   RETURN VALUE
 *       Number of bytes available in the buffer, for any
subsequent call
 *       to any stream-parsing routine
 *

*****************************************************************
*********/
int prepare_bitstream_adif()
{
  int len, frame_len;
  len = 0;
  //frame_len = params->frame_length;
  len = (bitstream_count > BS_BUF_SIZE) ? BS_BUF_SIZE :
bitstream_count;
  bitstream_buf_index += len;
```

```
  bitstream_count     -= len;
  in_buf_done         += len;
  bytes_supplied      += len;

  return(len);
}
```

Appendix A Header-data extraction and Presentation to the decoder

Channel-to-speaker mapping can be specified as part of the header in two ways

1. Program Configuration elements (PCE's)
2. Channel Configuration parameter

PCE is an elaborate and flexible way of specifying mappings. A header might contain zero or more PCE's. Every PCE also contains data about sampling-frequency used.

Channel Configuration parameter can take values in the range 0-7. Each value specified indicates preset-mapping defined in the standard. (IS 13818-7, Table 3.1)

AACD_Block_Params supports both of the above options in a mutually exclusive manner, i.e. only one of the above options can be used at a given time. The following are the fields in this struct.

```
typedef struct
{
    AACD_INT32        num_pce;
    AACD_ProgConfig   *pce;

    AACD_INT32 ChannelConfig;
    AACD_INT32 SamplingFreqIndex;
    AACD_INT32 BitstreamType;
    AACD_INT32 BitRate;
    AACD_INT32 BufferFullness; /*No. of bits in encoder buffer after
                               encoding first raw_data_block */
    AACD_INT32 ProtectionAbsent;
    AACD_INT32 CrcCheck;
    AACD_INT32 frame_length;

#ifdef OLD_FORMAT_ADTS_HEADER
    AACD_INT32 Flush_LEN_EMPHASIS_Bits;
#endif

} AACD_Block_Params;
```

For every PCE that occurs in the header, there is an associated value called 'buffer_fullness', if (bitstream_type == 0). i.e., a constant-bitrate stream. This pair consisting of a PCE and an associated 'buffer_fullness' value are stored in the ProgConfig struct. If (bitstream_type != 0), then the 'buffer_fullness' field is of no meaning and hence it can have any arbitrary value. Frow now on, PCE and ProgConfig will be used interchangeably.

If the header has no PCE, then num_pce should be set to zero. Otherwise, all the ProgConfig elements in the header must be put in an array and a pointer to the first-item in the array should be assigned to pce and num_pce should indicate how many PCE's are in the array.

To support both types of specifying channel-to-speaker mapping, the following protocol is to be used.

If Channel Configuration is to be used, then num_pce should be set to zero. Except pce all other fields will be used.

If PCE is to be used, then num_pce must be positive. In this case $ChannelConfig$ will be not be used. Also, $BufferFullness$, $SamplingFreqIndex$ will be ignored, as these are available in ProgConfig struct itself. (The PCE contains $SamplingFreqIndex$ value).

If cyclic-redundancy-check (crc check) is not be done, then $protection_absent$ must be 1.

PsyTEL Encoder

The field $Flush_LEN_EMPHASIS_Bits$, can have two values : 0 or 1. Some encoders, while generating adts outputs, use the old format adts header which has a two bit field named $emphasis$. This field will not be present in compliant-encoders. If the application knows that the input file uses old-format adts headers, then the decoder library should be compiled after defining OLD_FORMAT_ADTS_HEADER. In particular, the PsyTEL encoder (a freely available encoder) used old format adts headers and in addition it does not perform a byte-alignment after header-parsing is over. This is an error and decoding will fail. To avoid this error, two bits have to be flushed everytime the decoder begins decoding a fresh raw_data_block. To summarize, if the encoder used old format adts header and if the stream if of type MPEG4 and if there is no byte-alignment at the end of every header, then set $Flush_LEN_EMPHASIS_Bits$ to 1, otherwise set it to zero. This is true for the PsyTEL encoder.

Note that, compliant encoders do not use old-format adts headers and neither do fail to do byte-alignment.

Struct ProgConfig

```
typedef struct
{
    AACD_INT32          profile;
    AACD_INT32          sampling_rate_idx;
    AACD_EleList        front;
    AACD_EleList        side;
    AACD_EleList        back;
    AACD_EleList        lfe;
    AACD_EleList        data;
    AACD_EleList        coupling;
    AACD_MIXdown        mono_mix;
    AACD_MIXdown        stereo_mix;
    AACD_MIXdown        matrix_mix;
    /* Ignore the comment field to minimize memory usage */
    AACD_INT8           comments[1];
    AACD_INT32          buffer_fullness;
    AACD_INT32          tag;
}   AACD_ProgConfig;
```

$profile$ - This should be set to 1 always (MPEG4 AAC Low Complexity profile)
(IS 13818-7, Table 2.1, Profiles)
$sampling_rate_idx$ - This is an index into an array of valid sampling-frequency values
(IS 13818-7, Section 3.1.1, sampling_frequency_index)

$buffer_fullness$ - No. of bits remaining in the encoder buffer after encoding the first raw_data_block in the frame.
tag - id of a PCE

Struct EleList

```
typedef struct
{
    AACD_INT32              num_ele;
    AACD_INT32              ele_is_cpe[(1 << LEN_TAG)];
    AACD_INT32              ele_tag[(1 << LEN_TAG)];
}   AACD_EleList;
```

A channel can map to any of front, side, back, lfe (low sampling frequency enhancements) speakers. For e.g, for the front-channel, num_ele indicates the number of front-channel elements, ele_is_cpe indicates if a front-channel element is a channel-pair or not and ele_tag defines a unique id for a front-channel element.

Struct MIXdown

```
typedef struct
{
    AACD_INT32              present;
    AACD_INT32              ele_tag;
    AACD_INT32              pseudo_enab;
}   AACD_MIXdown;
```

Consider stereo mix down.
$present$ - indicates if a strereo-mix down element is present
ele_tag - indicates the number of a specified channel pair element that is the stereo mixdown element
$pseudo_enab$ - this is defined only for matrix mix-down and indicates if pseudo-surround is enabled or not.

More details can be found in the standards (IS 13818-7, ISO/IEC 14496-3)

