ARM MPEG—Advanced Audio Coding Decoder

Version 1

Programmer's Guide

ARM

ARM DUI 0129A

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Change
October 1999	A	First release

Proprietary Notice

Contents
Programmer's Guide

Preface

This preface introduces the ARM *Moving Pictures Experts Group* (MPEG)—*Advanced Audio Coding* (AAC) Decoder. It contains the following sections:

About this book

This book is provided with the ARM MPEG—AAC Decoder. It describes the *Application Program Interface* (API) to the decoder library.

Intended audience

This book is written for programmers who want to integrate the ARM MPEG—AAC Decoder into an embedded system.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM MPEG—AAC Decoder.

Chapter 2 *ARM MPEG--AAC Decoder Types and Constants*

Read this chapter for a description of the types and constants used by the ARM MPEG—AAC Decoder.

Chapter 3 *ARM MPEG--AAC Decoder Functions*

Read this chapter for a description of the functions provided by the ARM MPEG—AAC Decoder.

Chapter 4 *Example program*

Read this chapter to see a complete example implementation of the ARM MPEG—AAC Decoder.

Typographical conventions

The following typographical conventions are used in this book:

bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate.
italic	Highlights special terminology, denotes internal cross-references, and citations.
`typewriter`	Denotes text that may be entered at the keyboard, such as commands, file and program names, and source code.
`type`writer	Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`typewriter bold`	Denotes language keywords when used outside example code.

Further reading

This section lists publications from third parties that provide additional information on developing the ARM MPEG—AAC Decoder.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

Other publications

For reference information relating to the ARM MPEG—AAC Decoder, please refer to the following:

- ISO/IEC 13818-4:1997, *Information technology—Generic coding of moving pictures and associated audio information Part 4. Conformance testing*.
- ISO/IEC 13818-7:1997, *Information Technology—Generic coding of moving pictures and associated audio information Part 7. Advanced Audio Coding (AAC)*.

Feedback

ARM Limited welcomes feedback on both the ARM MPEG—AAC Decoder and its documentation.

Feedback on the ARM MPEG—AAC Decoder

If you have any problems with the ARM MPEG—AAC Decoder, please contact your supplier. To help them provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

If you have any comments on this book, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter provides an overview of the ARM MPEG—AAC Decoder. It includes the following section:

- *About the MPEG—AAC* on page 1-2.

1.1 About the MPEG—AAC

The ARM 2.0.0.0 Channel Low Complexity Profile MPEG-2 AAC decoder is an optimized library, designed to efficiently decode mono or stereo Low Complexity Profile AAC bitstreams on the ARMv4 processor family.

Note

The ARM MPEG—AAC Decoder will be referred to as MPEG—AAC throughout this book.

MPEG—AAC is an ARM implementation of the AAC audio compression standard. This standard was specifically designed for generic multi-channel audio data. Unlike other MPEG audio standards, such as MPEG Audio Layer III (MP3), MPEG—AAC is not restricted by the necessity to maintain backward compatibility with previous MPEG standards and, as a result, is more efficient, while maintaining a high audio quality.

MPEG—AAC is compliant with *ISO/IEC 13818-7:1997*.

All output sampling frequencies, from 8kHz to 96kHz, and all valid input rates (including variable-rate) are supported.

Chapter 2
ARM MPEG—AAC Decoder Types and Constants

This chapter describes the types and constants used by MPEG—AAC. It contains the following sections:

2.1 Enumerations

This section describes the enumerations available in the MPEG—AAC library:

- *The bitstream format enumeration (tAAC_BitstreamFormat)*
- *The original/copy enumeration (tAAC_OriginalCopy)* on page 2-3
- *The bitstream type enumeration (tAAC_BitstreamType)* on page 2-3
- *The profile enumeration (tAAC_Profile)* on page 2-3
- *The sampling frequency enumeration (tAAC_Frequency)* on page 2-4
- *The error code enumeration (tAAC_ErrorCode)* on page 2-4.

2.1.1 The bitstream format enumeration (tAAC_BitstreamFormat)

This enumeration is used to represent the format of the bitstream:

```
typedef enum
{
  eAAC_UnknownFormat,      // Uninitialized value
  eAAC_ADIF,       // ADIF (raw bitstream with single header)
  eAAC_ADTS,       // ADTS (framed bitstream with frame headers)
  eAAC_RAW         // raw bitstream without header
}
tAAC_BitstreamFormat;
```

ISO/IEC 13838-7 defines two bitstream formats:

Audio Data Interchange Format (ADIF)

A single header followed by a stream of raw data blocks.

Audio Data Transport Stream (ADTS)

The bitstream is divided into frames, each consisting of a header followed by one or more raw data blocks.

Additionally, MPEG—AAC can be used to decode a raw bitstream with no header. In this case, you must set the sampling frequency when initializing the decoder (see *AAC_Initialise()* on page 3-2). The number of channels will be counted automatically while decoding the first data block.

2.1.2 The original/copy enumeration (tAAC_OriginalCopy)

This enumeration is used to represent the field in the bitstream header that indicates whether the bitstream is an original recording or a copy:

```
typedef enum
{
  eAAC_Copy,      // Copy of a recording
  eAAC_Original   // Original recording
}
tAAC_OriginalCopy;
```

2.1.3 The bitstream type enumeration (tAAC_BitstreamType)

This enumeration is used to represent the field in the bitstream header that indicates whether the input bitrate is constant or variable:

```
typedef enum
{
  eAAC_ConstantRate,  // Constant input bitrate
  eAAC_VariableRate   // Variable input bitrate
}
tAAC_BitstreamType;
```

2.1.4 The profile enumeration (tAAC_Profile)

This enumeration is used to represent the field in the bitstream header that indicates which profile is required to decode the bitstream:

```
typedef enum
{
  eAAC_MainProfile,  // Main Profile
  eAAC_LCProfile,    // Low Complexity Profile
  eAAC_SSRProfile    // Scalable Sampling Rate Profile
}
tAAC_Profile;
```

Note

The ARM MPEG-AAC Decoder currently supports only Low Complexity Profile bitstreams.

2.1.5 The sampling frequency enumeration (tAAC_Frequency)

This enumeration is used to represent the sampling frequency of the bitstream. The numerical values in the enumerator names represent the sampling frequency in *Hertz* (Hz):

```
typedef enum
{
  eAAC_96000,
  eAAC_88200,
  eAAC_64000,
  eAAC_48000,
  eAAC_44100,
  eAAC_32000,
  eAAC_24000,
  eAAC_22050,
  eAAC_16000,
  eAAC_12000,
  eAAC_11025,
  eAAC_8000
}
tAAC_Frequency;
```

2.1.6 The error code enumeration (tAAC_ErrorCode)

This enumeration is used as the return value of all the MPEG—AAC functions to indicate whether the function completed successfully:

```
typedef enum
{
  eAAC_NoError,
  eAAC_EndOfBitstream,
  eAAC_HeaderMissing,
  eAAC_UnsupportedFeature,
  eAAC_TooManyChannels,
  eAAC_UnknownError
}
tAAC_ErrorCode;
```

where the error codes mean:

`eAAC_NoError`
: The function completed successfully.

`eAAC_EndOfBitstream`
: The function completed successfully, and all the valid data in the input buffer was used.

`eAAC_HeaderMissing`

The bitstream does not begin with either an ADIF or an ADTS header. If it is a valid raw bitstream, it can still be decoded. However, you must set the sampling frequency to the correct value when initializing the decoder (see *AAC_Initialise()* on page 3-2).

`eAAC_UnsupportedFeature`

The bitstream appears to use features which are not supported by the 2.0.0.0 Channel Low Complexity Profile.

`eAAC_TooManyChannels`

Either the bitstream contains more channels than MPEG—AAC will support, or the application has not provided enough output buffers for all the channels.

`eAAC_UnknownError`

Indicates that either the bitstream is invalid, or that an internal error has occurred.

Note

Because of the highly compressed nature of the bitstream, either of these `eAAC_UnknownError` problems could produce any of the error codes.

2.2 The bitstream structure

This structure is used to control and access the buffer that is used to transfer the audio bitstream to the decoder.

Example 2-1 The bitstream structure

```
typedef struct
{
  /* The following are maintained by the decoder. */
  /* They should not be altered by the application. */
  int             ValidBits;   /* For internal use  */
  unsigned long   CurrentWord; /* For internal use  */
  unsigned char  *DataOut;     /* Pointer to start of */
                               /* valid data  */

  /* The following are maintained by the application. */
  /* They will not be altered by the decoder. */
  unsigned long  *DataEnd;    /* Pointer to end of buffer */
  unsigned long  *DataStart;  /* Pointer to start of buffer */
  unsigned long  *DataIn;     /* Pointer to end of valid data */
}
tAAC_Bitstream;
```

The bitstream is held in a circular buffer, delimited by the `DataStart` and `DataEnd` pointers. The `DataStart` pointer must reference the first word of the buffer, and the `DataEnd` pointer must reference the first word after the end of the buffer.

You must place new data in the buffer at the position referenced by the `DataIn` pointer, which must be initialized to reference the first word of the buffer. You must update the `DataIn` pointer to reference the word after the last word that contains valid data. If the last word containing valid data is also the last word in the buffer, you must update the `DataIn` pointer to reference the first word of the buffer, and not the word after the end of the buffer. That is, you must update the `DataIn` pointer to reference the next location in which data will be placed.

Within each word, you must order the bits so that the most significant bit represents the first bit of the bitstream. For example, the bitstream fragment

```
0100 0001 0100 0100 0100 1001 0100 0110
```

would be represented by `0x41444946`.

The `DataOut` pointer references the next word that will be read by the decoder. When placing new data in the bitstream, you must not overwrite this word. You can place data in the word before this one.

You must ensure that the buffer contains at least 6144 bits (768 bytes) of valid data per audio channel when the decoder is invoked. This is the largest possible size of a data block.

You must keep the `DataStart`, `DataEnd`, and `DataIn` pointers aligned on 4-byte boundaries at all times. You can assume that the `DataOut` pointer will also be aligned on a 4-byte boundary.

Some applications might require that you use a linear buffer rather than a circular buffer. To use a linear buffer:

1. Initialize the `DataStart` pointer to reference the first word of the buffer, as described above.
2. Initialize the `DataEnd` pointer to `NULL` to indicate that there is no end to the buffer (the end of the valid data is marked by the `DataIn` pointer).
3. Place new data in the buffer, updating the `DataIn` pointer to reference the word after the last word containing valid data, as described above.
4. If, after decoding a block, the buffer contains less than 6144 bits of valid data per audio channel, you must copy the remaining data to the start of the buffer to add more data. The valid data begins with the word referenced by the `DataOut` pointer, and ends with the word before that referenced by the `DataIn` pointer. The `DataOut` and `DataIn` pointers must be updated to reference the first word of the buffer, and the word after the last word containing valid data, respectively.

2.3 The bitstream header structure

This structure is used to represent the information supplied in the header of a bitstream. Most of the information is not required by the decoder, and is provided for the benefit of the application. An exception to this is the sampling frequency field, where an incorrect value could result in distorted audio output.

If the bitstream begins with a valid ADIF or ADTS header, the first call to `AAC_DecodeHeader()` will fill all the relevant fields with valid values (see *AAC_DecodeHeader()* on page 3-3). Otherwise, the first call to `AAC_DecodeDataBlock()` will fill in the field for the number of channels, `NumberOfChannels`. All other fields are considered invalid.

All the fields are defined in *ISO/IEC 13818-7:1997, subclause 8.1.1*.

Example 2-2 The bitstream header structure

```
typedef struct
{
  tAAC_BitstreamFormat  BitstreamFormat;       // ADIF, ADTS or raw data
  char                  CopyrightID[9];        // 72-bit copyright ID
  int                   CopyrightBits;         // Number of currently valid bits
  tAAC_OriginalCopy     OriginalCopy;          // Original recording or copy
  tAAC_BitstreamType    BitstreamType;         // Variable or fixed rate
  int                   Home;
  int                   BitRate;               // Input rate; zero if unknown
  int                   BufferFullness;
  int                   FrameLength;           // Only for ADTS bitstreams
  int                   BlocksInFrame;         // Only for ADTS bitstreams
  int                   NumberOfChannels;      // Zero if not known
  int                   NumberOfFrontChannels; // Zero if not known
  int                   NumberOfSideChannels;  // Zero if not known
  int                   NumberOfBackChannels;  // Zero if not known
  int                   NumberOfLFEChannels;   // Zero if not known
  tAAC_Profile          Profile;               // Main, LC or SSR profile
  tAAC_Frequency        Frequency;             // Output sampling frequency
  char                  Comment[257];          // Null-terminated string
}
tAAC_Header;
```

2.4 The decoder type

MPEG—AAC requires a certain amount of RAM to function. For convenience, all of the required RAM (except for a small amount of stack usage) is collected into a single structure. This allows:

- several independent instances of the decoder to be used concurrently
- the memory to be easily reused by other applications when the decoder is not required.

This structure is regarded as an opaque type. A pointer of this type is passed to each of the MPEG—AAC functions. The type is declared as:

```
typedef struct sAAC_Decoder tAAC_Decoder;
```

The size is defined by a constant symbol which is exported from the library, and is declared as:

```
extern const int AAC_DecoderSize;
```


Chapter 3
ARM MPEG—AAC Decoder Functions

This chapter describes the functions provided by MPEG—AAC. It contains the following section:

- *MPEG—AAC functions* on page 3-2.

3.1 MPEG—AAC functions

The section describes each of the functions provided with the MPEG—AAC library:

- *AAC_Initialise()*
- *AAC_DecodeHeader()* on page 3-3
- *AAC_DecodeDataBlock()* on page 3-4.

3.1.1 AAC_Initialise()

This function sets up the decoder workspace, making it ready to decode a bitstream. This function must be called before the first frame of each new bitstream. If it is not called, there might be loud clicks in the output of the first block.

Syntax

```
tAAC_ErrorCode AAC_Initialise(tAAC_Decoder *Decoder,
                              tAAC_Bitstream *Bitstream,
                              tAAC_Header *Header,
                              tAAC_Frequency DefaultFrequency)
```

where:

Decoder Is the workspace to be used by the decoder. This will be initialized (see *The decoder type* on page 2-8).

Bitstream Is the incoming bitstream buffer. The `DataStart` and `DataEnd` pointers must point to the first byte of the buffer and the first byte above the buffer, respectively, before calling the function. The internal fields (`CurrentWord`, `ValidBits`, and `DataOut`) will be initialized by the function (see *The bitstream structure* on page 2-6).

Header Is the structure that will be used to record the bitstream header. All of its fields will be cleared (see *The bitstream header structure* on page 2-8). If you do not require the header information, you can pass a NULL pointer instead.

DefaultFrequency

Is the sampling frequency to use if the bitstream has no header. The decoder might fail to decode a raw bitstream if *DefaultFrequency* is incorrectly set (see *The sampling frequency enumeration—tAAC_Frequency* on page 2-4).

Return value

See *The error code enumeration (tAAC_ErrorCode)* on page 2-4.

3.1.2 AAC_DecodeHeader()

This function decodes the information contained in an ADIF or ADTS header.

Syntax

```
tAAC_ErrorCode AAC_DecodeHeader(tAAC_Decoder *Decoder,
                                tAAC_Bitstream *Bitstream,
                                tAAC_Header *Header)
```

where:

Decoder Is a pointer to the block of memory defined as your workspace (see *The decoder type* on page 2-8).

Bitstream Is the incoming bitstream buffer. On output, the `DataOut` pointer will be updated (see *The bitstream structure* on page 2-6).

Header Is a structure that will be used to record the bitstream header. On output, the fields will be filled in if a header is found (see *The bitstream header structure* on page 2-7).

Return value

See *The error code enumeration (tAAC_ErrorCode)* on page 2-4.

Notes

It is safe to call this function before each data block. If no header is found, the `AAC_DecodeHeader` function will have no effect. It is not necessary to call this function if you do not require the header information.

If the bitstream is in the ADIF format, you can call this function before the first data block. It is not necessary to call it more than once.

If the bitstream is in the ADTS format, you can call this function before each ADTS frame. The `BlocksInFrame` field will then indicate how many blocks you can decode before you call this function again. If you require the *copyright ID* field of the header, you must call this function before every ADTS frame because this information is spread over many frames.

If the bitstream has no header, none of the fields of the header are valid. The `AAC_DecodeDataBlock()` function will fill in the `NumberOfChannels` field based on the number of channels it encounters (see *AAC_DecodeDataBlock()* on page 3-4).

3.1.3 AAC_DecodeDataBlock()

This function decodes a block of data from the bitstream to produce 1024 new audio samples for each channel.

Syntax

```
tAAC_ErrorCode AAC_DecodeDataBlock(tAAC_Decoder *Decoder,
                                   tAAC_Bitstream *Bitstream,
                                   tAAC_Header *Header,
                                   short *Channels[])
```

where:

Decoder Is a pointer to the block of memory defined as your workspace (see *The decoder type* on page 2-8).

Bitstream Is the incoming bitstream buffer. On output, the `DataOut` pointer is updated to point to the start of the next block (see *The bitstream structure* on page 2-6).

Header Is the header structure for this bitstream. On output, more fields can be filled in if a *Program Configuration Element* (PCE) is found. The `NumberOfChannels` field will be filled in if set to zero on entry (see *The bitstream header structure* on page 2-8). If you do not require the header information, you can pass a NULL pointer instead.

Channels`[]` Is an array of pointers to output buffers. Each will be filled with 1024 16-bit PCM samples, decoded from the bitstream. You must terminate this list with a NULL pointer to ensure that the decoder does not attempt to decode additional channels for which there is no memory space allocation.

Return value

See *The error code enumeration (tAAC_ErrorCode)* on page 2-4.

Caution

The bitstream structure will be invalid if this function returns any value other than `eAAC_NoError`, and any attempt to continue decoding the bitstream will have unpredictable results.

Chapter 4
Example Program

This chapter provides an example implementation of MPEG_AAC. It contains the following section:

- *Example program* on page 4-2.

4.1 Example program

Example 4-1 is an example implementation of MPEG—AAC.

Example 4-1 Example implementation—AAC_Decoder.c

```
/*
 * AAC_Decoder.c
 * Example implementation of the ARM MPEG AAC Decoder.
 *
 * Copyright ARM Limited 1999. All Rights Reserved.
 *
 */

#include <stdio.h>

#include "AAC.h"

#define WORKSPACE_SIZE  16436
#define INBUFFER_SIZE     384  /* 2 channels require 12288 bits = 384 words */

/* Memory for the decoder's workspace */
static char          sWorkspace[WORKSPACE_SIZE];

/* Structure to record the bitstream header (optional) */
static tAAC_Header   sHeader;

/* The buffer for the input bitstream */
static unsigned long  InBuffer  [INBUFFER_SIZE];
static tAAC_Bitstream sBitstream =
{
  0, 0, 0, /* internal data will be initialised by AAC_Initialise() */
  InBuffer + INBUFFER_SIZE,    /* DataEnd   */
  InBuffer,                    /* DataStart */
  InBuffer                     /* DataIn    */
};

/* The audio output buffers */
static short         OutBuffer1[1024];
static short         OutBuffer2[1024];
static short        *OutBuffers[] = {OutBuffer1, OutBuffer2, NULL};

/* Function to read input data into the bitstream buffer (defined below)  */
static void FillBitstream(tAAC_Bitstream *Bitstream, FILE *File);
```



```
/* External function required to play or otherwise process the audio data */
extern void ProcessAudioData(short *Data[], int NumberOfChannels);

static const char *ErrorStrings[] =
{
  "No error", "End of bitstream", "Header missing",
  "Unsupported feature", "Too many channels", "Unknown error"
};

int main(int argc, char *argv[])
{
  tAAC_Decoder   *Decoder   = (tAAC_Decoder *)sWorkspace;
  tAAC_Bitstream *Bitstream = &sBitstream;
  tAAC_Header    *Header    = &sHeader;
  FILE           *InFile;

  /* Check that the workspace is large enough */
  if (sizeof(sWorkspace) < AAC_DecoderSize)
  {
  printf("Error: Workspace must be at least %d bytes.\n", AAC_DecoderSize);
  return 1;
  }

  /* Open the input file */
  InFile = fopen(argv[1], "rb");
  if (!InFile)
  {
    printf("Error: Unable to open '%s'.\n", argv[1]);
    return 1;
  }

  /* Initialise the decoder */
  if (AAC_Initialise(Decoder, Bitstream, Header, eAAC_44100) != eAAC_NoError)
  {
    printf("Error: Initialisation failed.\n");
    return 1;
  }

  /* Ensure there will be enough data available */
  FillBitstream(Bitstream, InFile);

  /* Read the header (if there is one) */
  AAC_DecodeHeader(Decoder, Bitstream, Header);
```

```
  for(;;)
  {
    tAAC_ErrorCode Error;
    /* Ensure there will be enough data available */
    FillBitstream(Bitstream, InFile);

    /* Extract the audio data */
    Error = AAC_DecodeDataBlock(Decoder, Bitstream, Header, OutBuffers);

    if (Error != eAAC_NoError && Error != eAAC_EndOfBitstream)
    {
      printf("Error: %s.\n", ErrorStrings[Error]);
      return 1;
    }

    /* call external function to process the audio data */
    ProcessAudioData(OutBuffers, Header->NumberOfChannels);

    if (Error == eAAC_EndOfBitstream)
    {
      printf("End of bitstream.\n");
      break;
    }
  }
  return 0;
}

/* Function to read input data into the bitstream buffer */
static void FillBitstream(tAAC_Bitstream *Bitstream, FILE *File)
{
  do
  {
    unsigned long  NewData;

    /* Read 32 bits from the bitstream, with the first bits on the right */
    /* Return immediately if there is no data available.                 */
    if ( (NewData = getc(File)) == EOF)
      return;

    NewData <<= 24;
    NewData |= (getc(File) & 0xff) << 16;
    NewData |= (getc(File) & 0xff) << 8;
    NewData |= (getc(File) & 0xff);

    /* Write this into the buffer */
    *Bitstream->DataIn = NewData;
```



```
    /* Update the buffer pointer */
    Bitstream->DataIn++;
    if (Bitstream->DataIn == Bitstream->DataEnd)
      Bitstream->DataIn = Bitstream->DataStart;
  } while( !feof(File) &&
    Bitstream->DataIn != Bitstream->DataOut );
}
```


Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

F

I

L

N

O

P

R

S

T

Numerics

