06-3644-API-TX30
AUG 06 , 2008
2.6

Application Programmers Interface for MP3 Decoder

ABSTRACT:

Application Programmers Interface for MP3 Decoder

KEYWORDS:

Multimedia codecs, MP3

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	18-Feb-2004	Zakir	Initial Draft
1.0	23-Feb-2004	Zakir	Internal review comments incorporated
1.1	06-Mar-2004	Zakir	Customers comments incorporated
1.2	21-Jun-2004	Zakir	Release for ARM9E
1.3	27-Jul-2004	Zakir	PCS comments incorporated.
2.0	06-Feb-2006	Lauren Post	Using new format
2.1	16-Oct-2006	Madhav Varma	Added bit-rate variable in the config structure
2.2	11-Apr-2007	Katherine Lu	Modified input buffer from 16-bit unit to 8-bit unit Updated structure to be same as the source code
2.3	17-Oct-2007	Terry Lv	Release of mp3d 1.08.
2.4	25-Mar-2008	Zhongsong Lin	Enable Push mode
2.5	20-May-2008	Bing Song	Add version information
2.6	06-Aug-2008	Baofeng Tian	Output data format change

Table of Contents

Introduction

1.1 Purpose

This document gives the details of the application programmer's interface of the MP3 Decoder.

1.2 Scope

This document describes only the functional interface of the MP3 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 reader is expected to have basic understanding of Audio Signal processing and MP3 decoding. The intended audience for this document is the development community who wish to use the MP3 decoder in their systems.

1.4 References

1.4.1 Standards

- ISO/IEC 11172-3:1993 Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s -- Part 3: Audio (popularly known as *MPEG-1 Audio*).
- ISO/IEC 11172-4:1995 Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s -- Part 4: Conformance testing (known as *MPEG-1 Conformance Testing*).
- ISO/IEC 13818-3:1998 Information technology -- Coding of moving pictures and associated audio information -- Part 3: Audio (popularly known as *MPEG-2 Audio LSF*).
- "MPEG Layer-3 Bitstream Syntax and Decoding" – by Ralph Sperschneider, issue – 1.3 dated 9th Sep 1997. (This document describes the bitstream syntax of of ISO/MPEG layer3 bitstream. This also includes syntax extension of MPEG Layer3 bitstream to meet the requirements of very low bitrates and sampling frequencies {8Khz, 11.025Khz, 12Khz}. These lower sampling frequencies syntax extension is not standardized by ISO is called MPEG-2.5)

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.

- J.P.Princen, A.W.Johnson, A.B.Bradley, “Subband/transform coding using filterbank design based on time domain aliasing cancellation”, in proc. IEEE Int. conference ASSP, april1987
- MPEG Layer3 Bitstream syntax and decoding (Includes MPEG 2.5 layer3)
- “A Tutorial on MPEG/Audio compression” by Davis Pan

1.4.3 Freescale Multimedia References

- MP3 Decoder Application Programming Interface – mp3_dec_api.doc
- MP3 Decoder Requirements Book - mp3_dec_reqb.doc
- MP3 Decoder Test Plan - mp3_dec_test_plan.doc
- MP3 Decoder Release notes - mp3_dec_release_notes.doc
- MP3 Decoder Test Results – mp3_dec_test_results.doc
- MP3 Decoder Performance Results – mp3_dec_perf_results.doc
- MP3 Decoder Interface Header – mp3_dec_interface.h
- MP3 Decoder Application Code – mp3_dec_api.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
DAC	Digital to Analog Converter
FSL	Freescale
IEC	International Electro-technical Commission
ISO	International Standards Organization
LC	Low Complexity
MDCT	Modified Discrete Cosine Transform
MP3	Layer 3, MPEG2 Layer3 and MPEG2.5 Layer 3
MPEG	Moving Pictures Expert Group
OS	Operating System
PCM	Pulse Code Modulation
PNS	Perceptual Noise Substitution
RVDS	ARM RealView Development Suite
TBD	To Be Determined
UNIX	Linux PC x/86 C-reference binaries

1.6 Document Location

docs/mp3_dec

2 API Description

This section describes the steps followed by the application to call the MP3 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 structure are prefixed as mp3d_ or app_ to indicate if that member variable needs to be initialized by the decoder or application.
The MP3 decoder API currently support 2 kinds of bitstream mode: the PUSH mode and the PULL (which has been adopted before this version) mode. In the PUSH mode, the decoder will not call swap functions to read in MP3 bitstreams as the original PULL mode does. It is the application's duty to supply enough bitstream for the decoder to decode one frame. PUSH and PULL mode is switched in the makefile of decoder, but both modes share the same interface definition.

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 {
      MP3D_Mem_Alloc_Info      mp3d_mem_info;
      Void                     *mp3d_decode_info_struct_ptr;
      MP3D_OUTPUT_FORMAT       app_out_format;
      MP3D_INT8                (*app_swap_buf) (
                                     MP3D_UINT8 ** new_buf_ptr,
                                     MP3D_INT32 *new_buf_len,
                                     MP3D_Decode_Config *dec_config);
      MP3D_INT8*                pInBuf;
      MP3D_INT16                inBufLen;
      MP3D_INT16                consumedBufLen;
} MP3D_Decode_Config;
```

Description of the decoder parameter structure

mp3d_mem_info

This is memory information structure. The application needs to call the function mp3d_query_dec_mem to get the memory requirements from decoder. The decoder will fill this structure. This will be discussed in step 2.

mp3d_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.

app_out_format

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 24bit samples.

app_swap_buf (used in PULL mode only)

Function pointer to swap buffers. The application has to initialize this pointer.

pInBuf (used in PUSH mode only)

Pointer. to input bitsream. The application has to set this value correctly before decoding one frame.

inBufLen (used in PUSHmode only)

Length of the input buffer..The application has to set this value correctly before decoding one frame.

consumedBufLen (used in PUSH mode only)

The length of bitstream consumed by decoding one frame. The decoder return this value to the application.

Example pseudo code for this step:

```
/* Allocate memory for the decoder parameter */
   dec_config = (MP3D_Decode_Config *)
                               alloc (sizeof(MP3D_Decode_Config);

/* Request the output PCM samples to be in 16bit format/24 bits */
   dec_config->app_output_precision = MP3D_16_BIT_OUTPUT;

#ifndef PUSH_MODE
/* Assign swap-buffer function to swap buffer function pointer */
   dec_config->app_swap_buffer = app_swap_buffers_mp3_dec;
#else
   dec_config->pInBuf = mp3_input;
   dec_config->inBufLen = MP3D_INPUT_BUF_SIZE;
#endif
```

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 *mp3d_decode_versionInfo* is :

C prototype:

```
const MP3D_INT8 * mp3d_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 MP3 Decoder Version Info
      printf("%s \n", mp3d_decode_versionInfo());
    }
```

Step 3: Get the decoder memory requirements

The MP3 decoder does not do any dynamic memory allocation. The application calls the function *mp3d_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 *mp3d_query_dec_mem* is :

C prototype:

```
MP3D_RET_TYPE mp3d_query_dec_mem (MP3D_Decode_Config * dec_config);
```

Arguments:

• dec_config	- Decoder config pointer.

Return value:

• MP3D_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:

<u>Memory information structure array</u>

```
typedef struct {
      /*  Number of valid memory requests */
      MP3D_INT32                  mp3d_num_reqs;
      MP3D_Mem_Alloc_Info_Sub           mem_info_sub[MAX_NUM_MEM_REQS];
} MP3D_Mem_Alloc_Info;
```

Description of the structure *MP3D_Mem_Alloc_Info*

<u>mp3d_num_reqs</u>

The number of memory chunks requested by the decoder.

<u>mem_info_sub</u>

This structure contains each chunk's memory configuration parameters.

```
typedef struct {
      MP3D_INT32  mp3d_size;          /* Size in bytes */
      MP3D_INT32  mp3d_type;          /* Memory type Fast or Slow */
      MP3D_MEM_DESC mp3d_mem_desc;
            /* Flag to indicate Static / Scratch memory */
      MP3D_INT32  mp3d_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*/
} MP3D_Mem_Alloc_Info_Sub
```

Description of the structure *MP3D_Mem_Alloc_Info_sub*

mp3d_size

The size of each chunk in bytes.

mp3d_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 request for a FAST_MEMORY for which the application allocates a SLOW_MEMORY, the decoder will still decode, but the performance (Mhz) will suffer.)

mp3d_mem_desc

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

mp3d_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 assign the base pointer of this chunk of memory to *app_base_ptr*. The application should allocate the memory which is aligned to a 4 byte boundary in any case.

Example pseudo code for the memory information request

```
/* Query for memory */
retval = mp3d_query_dec_mem (&dec_config);

if (retval != MP3D_OK)
      return 1;
```

Step 4: Allocate Data Memory for the decoder

In this step the application allocates the memory as required by MP3 Decoder and fills up the base memory pointer *‘app_base_ptr’ of ‘MP3D_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

```
MP3D_Mem_Alloc_Info_sub *mem;
```

```
/* Number of memory chunks requested by the decoder */
nr = dec_config->mp3d_mem_info.mp3d_num_req;

for(i = 0; i < nr; i++)
{
      mem = &(dec_config->mem_info_sub[i]);
      if (mem->mp3d_type == MP3D_FAST_MEMORY)
      {
            /* This function allocates memory in internal memory */
            mem->app_base_ptr = alloc_fast(mem->mp3d_size);
      }
      else
      {
            /* This function allocates memory in external memory */
            mem->app_base_ptr = alloc_slow(mem->mp3d_size);
      }
}
```

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

Step 5: Memory allocation for input buffer (pull mode)

The application has to allocate the memory needed for the input buffer. It is desirable to have the input buffer allocated in FAST_MEMORY, as this may improve the performance (Mhz) of the decoder. There is no restriction on the size of the input buffer to be given to the decoder. The recommended minimum size would be 2Kbytes. The decoder, whenever it needs the MP3 bit-stream, shall call the function *app_swap_buffers_mp3_dec* internally from the function *mp3_decode_frame.*
app_swap_buffers_mp3_dec should be implemented by the application. The application might have different techniques to implement this function. Sample code is given in section 5.1.1

Example pseudo code for allocating the input buffer

```
/* Allocate memory for input buffer */
input_buffer = alloc_fast(MP3D_INPUT_BUFFER_SIZE);
```

Step 6: Initialization routine

All initializations required for the decoder are done in *mp3d_decode_init*. This function must be called before the main decoder function is called. The input buffer pointer and the input buffer length needs to be passed to the initialization function. This is required by the decoder to start decoding the bitstream to begin with.

C prototype:

```
MP3D_RET_TYPE mp3d_decode_init (
                  MP3D_Decode_Config *,
                  MP3D_UINT8 *input buffer,
                  MP3D_INT32 input_buffer_length);
```

Arguments:

• Decoder parameter structure pointer.	
• input_buffer	Initial pointer to the input buffer
• input_buffer_length	Length of the input buffer passed in bytes.

Return value:

• MP3D_OK	-	Initialization successful.
• Other codes	-	Initialization Error

<u>Example pseudo code for calling the initialization routine of the decoder</u>

```
/* input buffer length to be passed */
input_buffer_length = MP3D_INPUT_BUFFER_SIZE;

/* Initialize the MP3 decoder. */
retval = mp3d_decode_init (dec_config, input_buffer,
input_buffer_length);

if (retval != MP3D_OK)
      return 1;
```

Step 7: 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 mp3d_decode_frame function. The application can allocate memory for output buffer using alloc_fast. Allocating memory in internal memory using alloc_fast will improve the performance (Mhz) of the decoder marginally. Mp3 decoder provide two type of output data format, correspondingly, there should exist two kind of output buffer allocation case.

<u>Example pseudo code for allocating memory for output buffer</u>

```
/* Allocate memory for output buffer */
MP3D_FRAME_SIZE = 576;
if( dec_config->app_out_format == MP3D_24_BIT_OUTPUT )
  outbuf = (MP3D_INT32 *)alloc_fast ((sizeof(long))*2*2*MP3D_FRAME_SIZE);
else
 outbuf = (MP3D_INT32 *)alloc_fast ((sizeof(short))*2*2*MP3D_FRAME_SIZE);
```

Step 8: Call the frame decode routine

The main MP3 decoder function is *mp3d_decode_frame*. This function decodes the MP3 bit stream in the input buffer to generate one frame of decoder output per channel in every call. The output buffer is filled with left channel and the right channel samples interleaved. For 16-bit stereo output, left sample and right sample use one compact 32-bit word. For 24-bit stereo output, each sample use one 32-bit word, locate in lsb part. In case of mono streams, the decoder fills samples in a sequential order.

The decoder fills up the structure MP3D_Decode_Params.

```
typedef struct {
      MP3D_INT32  mp3d_sampling_freq;
            /* Sampling frequency of the current frame in Khz  */
      MP3D_INT32  mp3d_num_channels;
            /* Number of channels decoded in current frame */
      MP3D_INT32  mp3d_frame_size;
            /* Number of stereo samples being outputted for this frame */
      MP3D_INT32  mp3d_bit_rate;
            /* Indicates bit- rate in kbps .
              The decoder fills this variable after parsing the header*/
      MP3D_INT32 layer;
            /* MPEG layer of the current stream
             * 1 for layer-I,2 for layer-II and 3 for layer-III */
      MP3D_INT32 mp3d_remain_bytes;
} MP3D_Decode_Params;
```

If the bit stream has errors, the decoder will try to find the next sync pattern for all types of errors (mentioned in the appendix), except for the MP3D_END_OF_STREAM.

C prototype:

```
MP3D_RET_TYPE    mp3d_decode_frame (
                        MP3D_Decode_Config *dec_config,
                        MP3D_Decode_Params *dec param,
                        MP3D_INT32 *output_buffer);
```

Arguments:

• dec_config	Decoder parameter structure pointer
• dec_param	Decoder output parameter pointer
• output_buffer	Pointer to the output buffer to hold the decoded samples

Return value:

• MP3D_OK	Indicates decoding was successful.
• **Others**	**Indicates error**

When the decoder encounters the end of bitstream, 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 bitstreams, as mentioned earlier the decoder fills samples in a sequential order. 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.

<u>Example pseudo code for calling the main decode routine of the decoder</u>

```
#ifdef PUSH_MODE
/* a size thread is used to make sure one frame of mp3 bitstream is in
the input buffer when in PUSH mode
*/
   const int sizeThread = SIZE_THREAD;
```

```
#endif
while (TRUE)
{
      /* Decode one frame */
      /* The decoded parameters for this frame is available in the
       * structure MP3D_Decode_Params */
      retval = mp3d_decode_frame (dec_config, &dec_param, outbuf);


      if (retval == MP3D_END_OF_STREAM)
      {
            /* Reached the end of bitstream */
            break;
      }

      if (retval != MP3D_OK)
      {
            /* Invalid frame encountered, do not output */
            continue;
      }

      /* If mono, copy the left channel to the right channel. */
      if (dec_param.mp3d_num_channels == 1)
      {
            int i;
            i = dec_param.mp3d_frame_size-1;
            for (; i >= 0; i--)
            {
                outbuf [2 * i] = outbuf [i];
                outbuf [2 * i + 1] = 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
       * output by the application.
       */
       audio_output_frame(outbuf, 2*MP3D_FRAME_SIZE);

      #ifdef PUSH_MODE
      /*
        In PUSH mode, input buffer size should be large than SIZE_THREAD
        in the exception that the end of bitstream has been reached

      */

         leftLength = dec_config->inBufLen - dec_config->consumedBufLen;
         if(leftLength<0)
            break;

         if(leftLength>sizeThread)
        {
            dec_config->pInBuf += dec_config->consumedBufLen;
            dec_config->inBufLen -= dec_config->consumedBufLen;
            dec_config->consumedBufLen =0;
        }
       else
      {

            memcpy(mp3_input, (dec_config->pInBuf+dec_config-
>consumedBufLen),leftLength);
```

```
            leftInLength = MP3D_INPUT_BUF_SIZE - leftLength;
            if(leftInLength > uFileSize- InBufLen)
                              leftInLength = uFileSize- InBufLen;

           if(InBufLen<uFileSize)
memcpy(mp3_input+leftLength,(input_buffer+InBufLen),leftInLength);

           InBufLen += leftInLength;
           dec_config->pInBuf = mp3_input;
           dec_config->inBufLen = leftLength + leftInLength;
           if(dec_config->inBufLen<=0)
                               break;
      dec_config->consumedBufLen =0;
                        }
      #endif
}
```

Step 9: Free memory

The application releases the memory that it allocated to MP3 Decoder if it no longer needs the decoder instance.

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

2.1 Call back function usage

Call back function is only used when in PULL mode. It is called by the decoder to get a new input buffer for decoding. This function is called by the MP3 decoder within the *mp3d_decode_frame* function when it runs out of current bit stream input buffer. The decoder uses this function to return the used buffer and get a new bit stream buffer. The call back function call by the decoder will be a function pointer call. This function will be assigned to the pointer app_swap_buff before the init is called.

This function should be implemented by the application. The parameter new_buf_ptr is a double pointer. This will hold the recently used buffer by the decoder when this function is called. The application can decide to free this or do any sort of arithmetic to get any new address. The application needs to put the new input buffer pointer in *new_buf_ptr to be used by the decoder.

The interface for this function is described below:

C prototype:

```
MP3D_INT8 (* app_swap_buff) (
                  MP3D_UINT8 ** new_buf_ ptr,
                  MP3D_INT32  * new_buf_len,
                  MP3D_Decode_Config *mp3d_decoder_config);
```

Arguments:

• *new_buf_ ptr*	- Pointer to the new buffer given by the application.
• *new_buf_len*	- Pointer to length of the new buffer in bytes. (8bit)
• *mp3d_decoder_config*	- Decoder configuration structure.

Return value:

• 0	- Buffer allocation successful.
• -1	- End of bitstream

Example pseudo code

```
INT8 app_swap_buffers_mp3_dec (MP3D_UINT8 ** new_buf_ptr,
                               MP3D_INT32 * new_buf_len,
                               MP3D_Decode_Config *mp3d_decoder_config)
{
      Request for an input buffer from the application.

      Wait for the input buffer.

      If the new buffer arrives
      {
            Return the used buffer to the application.

            Set the new_buf_ptr to point to the new buffer and
            set *new_buf_len to the length of the new buffer.


            Return 0 to the calling function to indicate that new buffer
            has been received.
      }

      Else if the application indicates end of bit stream
      {
            Set new_buf_ptr to NULL and *new_buf_len to 0.

            Return -1 to the calling function to indicate the end of
            input bit stream.
      }
}
```