08-6338-API-ZCH66
JUN 27 , 2008
1.1

# Application Programmers Interface for MP3 encoder

**ABSTRACT:**

Application Programmers Interface for MP3 encoder

**KEYWORDS:**

Multimedia codecs, MP3

**APPROVED:**

Shang Shidong

# Revision History

| VERSION | DATE | AUTHOR | CHANGE DESCRIPTION |
|---|---|---|---|
| 0.1 | 24-Aug-2007 | Wang Qinling | Initial Draft |
| 0.2 | 5-Sep-2007 | Wang Qinling | Modified |
| 0.3 | 28-May-2008 | Huang Shen | Add MP3ECodecVersionInfo() |
| 1.0 | 27-June-2008 | Xu Lionel | Porting to arm12 |
| 1.1 | 29-Dec-2008 | Wang Shengjiu | ADD mono input |

# Table of Contents

# Introduction

## 1.1 Purpose

The purpose of this document is to describe the interfaces for the MP3 encoder on the ARM12 core. It provides:
a) the function level interfaces of the encoder,
b) an example calling sequence, and
c) a brief description of the resource requirements of the encoder.

## 1.2 Scope

This document describes only the functional interface of the MP3 encoder. It does not describe the internal design of the encoder. Specifically, it describes only those functions by which a software module can use the encoder.

## 1.3 Audience Description

The reader is expected to have basic understanding of Audio Signal processing and MP3 encoding. The intended audience for this document is the development community who wish to use the MP3 encoder 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*).

### 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
- Davis Pan, “A Tutorial on MPEG/Audio compression”

### 1.4.3 Freescale Multimedia References

- MP3 Encoder Application Programming Interface – mp3_enc_api.doc
- MP3 Encoder Requirements Book - mp3_enc_reqb.doc
- MP3 Encoder Test Plan - mp3_enc_test_plan.doc
- MP3 Encoder Release notes - mp3_enc_release_notes.doc
- MP3 Encoder Test Results – mp3_enc_test_results.doc
- MP3 Encoder Performance Results – mp3_enc_perf_results.doc
- MP3 Encoder Interface Header – mp3_enc_interface.h

## 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 | MPEG Layer 3, as defined by ISOIEC 111723 and 138183. |
| MPEG | Moving Pictures Expert Group |
| OS | Operating System |
| PCM | Pulse Code Modulation |
| PNS | Perceptual Noise Substitution |
| RVDS | ARM RealView Development Suite |

## 1.6 Document Location

docs/mp3_enc

# 2 API Description

This section describes the steps followed by the application to call the MP3 encoder. During each step the data structures used and the functions used will be explained. Code is given at the end of each step. The member variables inside the structure are prefixed as mp3e_ or app_ to indicate if that member variable needs to be initialized by the decoder or application.

Mp3 encoder support push mode input.

## Step 1: Allocate memory for encoder parameter structure

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

```
/* Encoder  parameter  structure */

typedef struct
{
        MP3E_INT32 instance_id;
        MP3E_Mem_Alloc_Info mem_info[ENC_NUM_MEM_BLOCKS];
        MP3E_INT32 num_bytes;

}MP3E_Encoder_Config;
```

**Description of the decoder parameter structure**

*instance_id*

This is an ID index of instance.

*mem_info*

This is memory information structure. The application needs to call the function mp3e_query_mem to get the memory requirements from encoder. The encoder will fill this structure. This will be discussed in step 2.

*num_bytes*

Output the final byte number encoded.

Example code for this step:

```
/* Allocate memory for the encoder parameter stucture */
MP3E_Encoder_Config *enc_config;

enc_config->mem_info[0].ptr = (int *) malloc (23052);
enc_config->mem_info[1].ptr = (int *) malloc (1700);
enc_config->mem_info[2].ptr = (int *) malloc (1036);
enc_config->mem_info[3].ptr = (int *) malloc (172);
enc_config->mem_info[4].ptr = (int *) malloc (1596);
enc_config->mem_info[5].ptr = (int *) malloc (452);
```

# Step 2: Get the encoder memory requirements

The MP3 encoder does not do any dynamic memory allocation. The application calls the function *mp3e_query_mem* to get the encoder memory requirements. This function must be called before all other encoder functions are invoked.

This function should be called for each instance of the encoder. Each call to this function gives the amount of dynamic memory required by the encoder for its buffers. The total memory required is divided into six different blocks. mp3e_query_mem returns the sizes of these six different memory blocks along with their type and alignment.

The function prototype of *mp3e_query_mem* is :

**C prototype:**

*MP3E_RET_VAL mp3e_query_mem (MP3E_Encoder_Config *enc_config);*

**Stucture:**

```
typedef struct
{
   MP3E_MEM_DESC type;    /* Memory block type (Fast or Slow) */
   MP3E_INT32 size;       /* Memory block size */
   MP3E_INT32 align;      /* Memory block alignment in bytes */
   MP3E_INT32 *ptr;       /* Memory block pointer */
}MP3E_Mem_Alloc_Info;
```

**Arguments:**

- enc_config -- Encoder config pointer.

**Description of the structure *MP3E_Mem_Alloc_Info***

*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.

*size:*

The size of each chunk in bytes.

*align*:

Memory block alignment in bytes

*ptr*

Memory block pointer, this parameter needs to be assigned by the application to the memory block of the given size that is allocated for this block. i.e., ptr points to the address of the allocated memory block.

Example code for the memory information request

```
/* Query for memory */
rflag = mp3e_query_mem (&enc_config);
```

```
if (rflag!= MP3E_SUCCESS)
      return 1;
```

**Return value:**

- MP3E_SUCCESS -- Query memory is successful.
- MP3E_ERROR_INIT_QUERY_MEM -- Query memory is unsuccessful.

# Step 3: Allocate Data Memory for the encoder

In this step the application allocates the memory as required by MP3 Encoder. The application must allocate six chunks of memory requested by the encoder.

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

```
MP3E_RET_VAL rflag = MP3E_SUCCESS;

 enc_config ->mem_info[0].type = FAST_STATIC_MEMORY;
 enc_config ->mem_info[0].size = 23052; /*size in Bytes*/
 /* requires maximum memory of 23052 bytes*/
 enc_config ->mem_info[0].align = 4;

 enc_config ->mem_info[1].type = FAST_STATIC_MEMORY;
 enc_config ->mem_info[1].size = 1700;
 /* requires maximum memory of 1700 bytes */
 enc_config ->mem_info[1].align = 4;

 enc_config ->mem_info[2].type = FAST_STATIC_MEMORY;
 enc_config ->mem_info[2].size = 1036;
 /* requires maximum memory of 1036 bytes */
 enc_config ->mem_info[2].align = 4;

 enc_config ->mem_info[3].type = FAST_STATIC_MEMORY;
 enc_config ->mem_info[3].size = 172;
 /* requires maximum memory of 0172 bytes */
 enc_config ->mem_info[3].align = 4;

 enc_config ->mem_info[4].type = FAST_STATIC_MEMORY;
 enc_config ->mem_info[4].size = 1596;
 /* requires maximum memory of 1596 bytes */
 enc_config ->mem_info[4].align = 4;

 enc_config ->mem_info[5].type = FAST_STATIC_MEMORY;
 enc_config ->mem_info[5].size = 452 ;
 /* requires maximum memory of 0452 bytes */
 enc_config ->mem_info[5].align = 4;

 return rflag;
```

# Step 4: Output the MP3 Encoder Version Info

A routine to report MP3 encoder version info is required.  It is called by application.

**C prototype:**

```
const char *MP3ECodecVersionInfo (void);
```

**Arguments:**
None

**Return value:**
A constand string is returned to report version info.

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

```
    // Output the MP3 Encoder Version Info
    printf("%s \n", MP3ECodecVersionInfo());
```

# Step 5: The description of input buffer

The application has to allocate memory 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 encoder. The size of this input buffer will be 2 times the MP3 encoder frame in words (16bits). The application passes the pointer of the input buffer address to MP3 encoder.

# Step 6: Initialization routine

All initializations required for the encoder are done in *mp3e_encode_init*. This function must be called before the main encoder function is called. The encoder parameter and encoder config need to be passed to the initialization function. This is required by the encoder to start encoding the bitstream to begin with.

This function initializes the various buffers and variables required by the MP3 encoder. This function needs to be called by the application before encoding every file. Many parameters such as bit rate, sampling rate etc.. need to be filled in the MP3E_Encoder_Parameter structure by the application before calling this function. The init function writes the minimum size of the output buffer needed in the element mp3e_outbuf_size of the MP3E_Encoder_Parameter structure. We will discuss the structure of MP3E_Encoder_Parameter in step 7.

**C prototype:**

```
MP3E_RET_VAL mp3e_encode_init (MP3E_Encoder_Parameter *params,
MP3E_Encoder_Config *enc_config);
```

**Arguments:**
encoder parameter structure pointer.

- *params* -- the pointer to the encoder p*arameter*
- *enc_config* -- the pointer to the encoder *Config*.

**Return value:**

- MP3E_*SUCCESS* -- Initialization successful.
- MP3E_ERROR_INIT_BITRATE

-- Initialization Error. If the bitrate passed by the application to the init routine is invalid

- MP3E_ERROR_INIT_SAMPLING_RATE

-- Initialization Error. If the sampling rate passed by the application to the init routine is invalid

- MP3E_ERROR_INIT_MODE

-- Initialization Error. If the stereo mode passed by the application to the init routine is invalid

- MP3E_ERROR_INIT_FORMAT

-- Initialization Error. If the input format passed by the application to the init routine is invalid

- MP3E_ERROR_INIT_QUALITY

-- Initialization Error. If the value of quality passed by the application to the init routine is invalid

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

```
MP3E_Encoder_Config enc_config;
MP3E_Encoder_Parameter params;      /* Structure to pass input parameters
                                        to the encoder */

params.app_sampling_rate = sfreq;   /* set sampling rate */
params.app_bit_rate = bitrate;      /* set bit rate */
params.app_mode = mode;             /* set mode */

val = mp3e_encode_init (&params,&enc_config);
if (val != MP3E_SUCCESS)
        return 1;
```

# Step 7: The description of output buffer

The application has to allocate memory for the output buffers to hold the encoded MP3 samples for one frame size. The pointer to this output buffer needs to be passed to the mp3e_encode_frame function. The output buffer size is fixed under the fixed bitrate and sample rate, and there are different output buffer sizes with different bitrates and sample rates. The maximum value that can be returned by the MP3 encoder for this output buffer size is 1440 bytes. Therefore, the application has to allocate memory for the output buffer after executing mp3e_encode_init function.

# Step 8: Call the frame encode routine

The main MP3 encoder function is *mp3e_encode_frame*. This is the main encoding function that encodes a frame of 16 bit stereo PCM samples. This function get the pointer of the output buffer address passed by the application This is due to the fact that encoding an input frame results in a variable number of bytes, which may or may not constitute a full output frame size.

The decoder fills up the structure MP3E_Encode_Params.

```
typedef struct{
      MP3E_INT32 app_sampling_rate;
      MP3E_INT32 app_bit_rate;
      MP3E_INT32 app_mode;
      MP3E_INT32 mp3e_outbuf_size;
}MP3E_Encoder_Parameter;.
```

**C prototype:**

```
void mp3e_encode_frame (MP3E_INT16 *inbuf, MP3E_Encoder_Config
*enc_config,MP3E_INT8 *outbuf);
```

**Arguments:**

- `inbuf` -- Pointer to the input buffer to hold the encoded samples
- enc_config -- Encoder config structure pointer
- outbuf -- Pointer to the output buffer

**Description of the structure `MP3E_Encoder_Parameter`**

`app_sampling_rate`:

Sampling rate of the input file in Hz. The following sampling rates are possible: 32000, 44100 and 48000. This parameter needs to be filled by the application.

`app_bit_rate`:

Bitrate for encoding, in kbps. The following bit rates are possible: 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320 kbps. This parameter needs to be filled by the application.

`app_mode`:

Mode for the encoder. The various modes are defined by different bit fields of this 32-bit word. This parameter needs to be filled by the application. The following bits are used:

b1-b0: Stereo mode bits Two values are currently possible:

00: stereo mode is joint stereo

01: stereo mode is mono

b9-b8: Input format bit

00: Input format is L/R interleaved

01: Input format is with contiguous L samples, followed by contiguous R samples

b17-b16: Input quality bits

00: Low quality

01: High quality

Other bits are reserved.

`mp3e_outbuf_size`:

Size of the required output buffer in bytes. The MP3 encoder will fill this parameter and return, the application has to allocate an output buffer of this size or more. The maximum value that can be returned by the MP3 encoder for this output buffer size is 1440 bytes.

Example code for calling mp3e_encode_frame routine of the encoder

```
#define NUM_SAMPLES 1152         /* 1152 samples per channel */
#define MAX_OUTPUT_SAMPLES 1440  /*maximum output number of per instance
supported*/

MP3E_INT16 inbuf[NUM_SAMPLES*2];
MP3E_INT8 outbuf[MAX_OUTPUT_SAMPLES];
mp3e_encode_frame (inbuf,&enc_config,outbuf);
```

# 3 Appendix

Example code for calling the main encode routine:

```
if(type == "wav")
    {
      /* If the input file is a wav file, use afread function to read the
        samples */
        if(num_channels == 1)              /*mono wave file*/
        {
            short int buffer[NUM_SAMPLES*2];
            int i,j;

            while ((samp_ret = afread (inbuf, sizeof (short int),
num_samples, ifp)) ==  num_samples)
            {
                encode_frame_mp3e (inbuf,&enc_config);
                /* This function does encoding of one frame. It
                   internally calls swap_output_buf_mp3e() when
                   one full MP3 output buffer is available. */
            }
            /* Fill out the last frame with zeros before passing
             * it to the encoder*/
            if(samp_ret>0 && samp_ret != num_samples)
            {
                for(i=0; i<((num_samples-samp_ret)*2); i++)
                    inbuf[samp_ret+i]=0;    /*Fill remaining part of
                                                 the frame with 0*/
                encode_frame_mp3e (buffer,&enc_config);
            }
        }
        else if (num_channels == 2)          /*stereo wave file*/
        {
            while ((samp_ret = afread (inbuf, sizeof (short int),
num_samples*2, ifp)) ==  num_samples*2)
            {

                encode_frame_mp3e (inbuf,&enc_config);
                /* This function does encoding of one frame. It
                   internally calls swap_output_buf_mp3e() when
                   one full MP3 output buffer is available. */
            }
            /* Fill out the last frame with zeros before passing it
             * to the encoder */
            if(samp_ret>0 && samp_ret != num_samples*2)
            {
                int i;
                for(i=0; i<(num_samples*2-samp_ret); i++)
                    inbuf[samp_ret+i]=0;
                encode_frame_mp3e (inbuf,&enc_config);
            }
        }
        else
            fprintf(stderr,"More than 2 channel input not supported\n");
    }
    else
    {
        /* If the input file is a pcm file, use fread function to read
the samples */
        while ((samp_ret = fread (inbuf, sizeof (short int),
num_samples*2, ifp)) ==  num_samples*2)
        {
            encode_frame_mp3e (inbuf,&enc_config);
            /* This function does encoding of one frame. It
               internally calls swap_output_buf_mp3e() when
               one full MP3 output buffer is available. */
        }
        /* Fill out the last frame with zeros before passing it to the
encoder */
        if(samp_ret>0 && samp_ret != num_samples*2)
        {
            int i;
            for(i=0; i<(num_samples*2-samp_ret); i++)
                inbuf[samp_ret+i]=0;
            encode_frame_mp3e (inbuf,&enc_config);
        }
    }
flush_bitstream_mp3e(&enc_config);   /* flush any pending
                                        output bytes in the encoder */
```