08-6291-API-ZCH66

10/15/2008

1.0

Application Programming Interface for WMA v8 Encoder

ABSTRACT:

Application Programming Interface for WMA v8 Encoder

KEYWORDS:

Multimedia codecs, WMA, Windows Media Audio

APPROVED:

Shang Shidong

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	14-Nov-2007	Yin Haiting	Initial Draft
1.0	10-Oct-2008	Haiting Yin	Update copyright

Table of Contents

1 Introduction

1.1 Purpose

This document gives the application programming interface for the WMA v8 Encoder.

1.2 Scope

This document describes only the functional interface of the WMA 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 WMA encoding. The intended audience for this document is the development community who wish to use the WMA Encoder in their systems.

1.4 References

1.4.1 Standards

- "An overview of Window Media Audio Encoding", Microsoft Corporation
- ASF Specification from Microsoft Corporation, Revision 01.20.02, June 2004
- WMA Decoding Profiles Microsoft Corporation
- WMA Audio Concepts Microsoft Corporation

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

1.4.3 Freescale Multimedia References

- WMA v8 Encoder Requirements Book – wma8_enc_reqb.doc
- WMA v8 Encoder Test Plan – wma8_enc_test_plan.doc
- WMA v8 Encoder Release notes – wma8_enc_release_notes.doc
- WMA v8 Encoder Test Results – wma8_enc_test_results.doc
- WMA v8 Encoder Interface header – wma8_enc_interface.h

- WMA v8 Encoder Application Code – wma_enc_test.c

1.5 Definitions, Acronyms, and Abbreviations

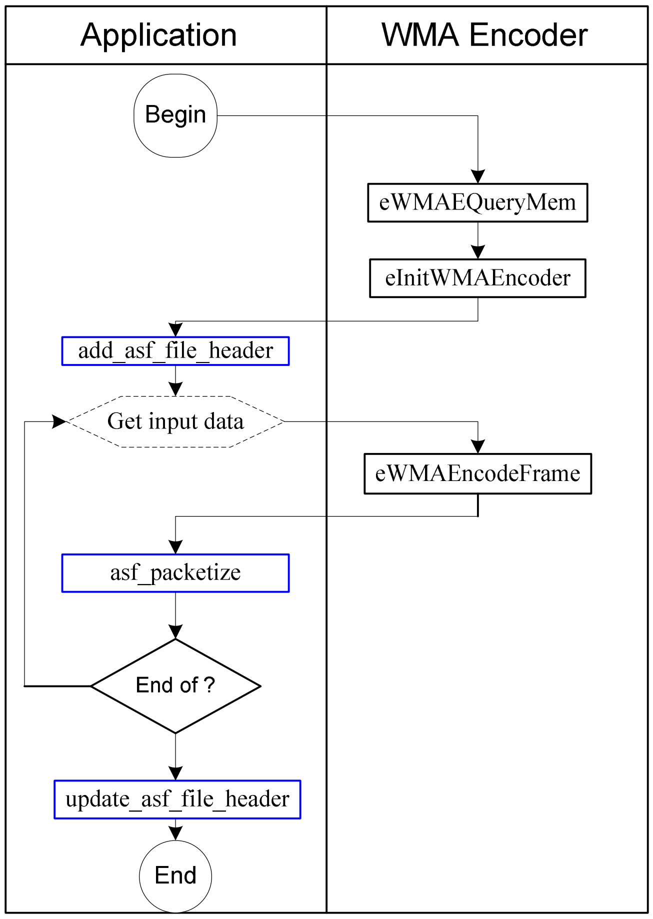
TERM/ACRONYM	DEFINITION
API	Application Programming Interface
ARM	Advanced RISC Machine
ASF	Advanced Streaming Format
DAC	Digital to Audio Converter
LC	Low Complexity
MLT	Modulated Lapped Transform
OS	Operating System
PCM	Pulse Code Modulation
WMA	Windows Media Audio

1.6 Document Location

docs/wma8_enc

2 Flow diagram of Interaction between an Application and WMA Encoder

The operations drawing in blue are for ASF packet use, if the output is raw WMA packet, these operations can be removed.



3 API Description

This section describes the steps followed by the application to call the WMA v8 Encoder. 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 WMAE_ or App_ to indicate if that member variable needs to be initialized by the encoder or application.

Step 1 (Optional): Get Version Information

Version information can be obtained by the following API:
const char * WMA8ECodecVersionInfo();

The format is defined as:

```
#define SEPARATOR " "

#define BASELINE_SHORT_NAME "WMA8E_ARM11_02.05.00"

#ifdef __WINCE
#define OS_NAME "_WINCE"
#else
#define OS_NAME ""
#endif

#ifdef DEMO_VERSION
#define CODEC_RELEASE_TYPE "_DEMO"
#else
#define CODEC_RELEASE_TYPE ""
#endif

/* user define suffix */
#define VERSION_STR_SUFFIX ""

#define CODEC_VERSION_STR \
    (BASELINE_SHORT_NAME OS_NAME CODEC_RELEASE_TYPE \
     SEPARATOR VERSION_STR_SUFFIX \
     SEPARATOR "build on" \
     SEPARATOR __DATE__ SEPARATOR __TIME__)
```

For instance:

```
"WMA8E_ARM11_02.05.00 build on May 14 2008 16:21:50"
```

is returned by calling WMA8ECodecVersionInfo(), meaning WMA8 encoder for ARM11, version 2.05.00.

Step 2: Allocate memory for encoder parameter structure

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

•

```
/* WMA Encoder configuration structure */
typedef struct
{
  WMAEMemAllocInfo            sWMAEMemInfo;   //Memory Info struct
  void                       *psWMAEncodeInfoStructPtr; // Global_struct
  WMAEEncoderParams          *psEncodeParams; //for Encoder Params
}WMAEEncoderConfig;
```

Description of the encoder parameter structure ***`WMAEEncoderConfig`***

sWMAEMemInfo

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

psWMAEncodeInfoStructPtr

This is a void pointer. This will be initialized by the encoder during the initialization routine. This will then be a used by the encoder and should not be changed by the application.

psEncodeParams: This is pointer to `WMAEEncoderParams` structure. Some member of this structure needs to be initialized by the application.

Step 3: Get the encoder memory requirements

The WMA8 Encoder does not do any dynamic memory allocation. The application calls the function eWMAEQueryMem to get the encoder memory requirements. This function must be called before all other encoder functions are invoked.

The function prototype of eWMAEQueryMem is:

C prototype:
```
tWMAEncodeStatus eWMAEQueryMem( WMAEEncoderConfig *psEncodeConfig);
```
Arguments:

- psEncodeConfig - Encoder configuration pointer.

Return value:

- WMA_Succeeded - 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 {
    WMAE_INT32               s32NumReqs;
    WMAEMemAllocInfoSub      sMemInfoSub [WMAE_MAX_NUM_MEM_REQS];
} WMAEMemAllocInfo;
```

Description of the structure *`WMADMemAllocInfo`*

s32NumReqs: The number of memory chunks requested by the encoder.

sMemInfoSub:

This structure contains each chunk's memory configuration parameters.

```
typedef struct {
    WMAE_INT32        s32WMAESize;      /* Size in bytes */
    WMAE_INT32        s32WMAEType;      /* Memory type Fast or Slow */
    WMAE_MEM_DESC     s32WMAEMemDesc;   /* to indicate if it is scratch
memory */
    WMAE_INT32        s32WMAEPriority;  /* In case of fast memory, specify
the priority */
    void              *app_base_ptr;    /* Pointer to the base memory ,
which will be allocated and
                                      * filled by the  application */
} WMAEMemAllocInfoSub;
```

Description of the structure *WMAEMemAllocInfoSub*

s32WMAESize

The size of each memory chunk will be in bytes.

s32WMAEType

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 encoder request for a FAST_MEMORY for which the application allocates a SLOW_MEMORY, the encoder will still encode, but the performance (MHz) will suffer.)

s32WMAEMemDesc

This indicates if the memory chunk is scratch memory

s32WMAEPriority

In case, if the encoder 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 is pointer to the base memory, which will be allocated and filled by application

Step 4: Allocate Data Memory for the encoder

In this step the application allocates the memory as required by WMA Encoder and fills up the base memory pointer 'app_base_ptr' *of* '*WMAEMemAllocInfoSub'* structure for each chunk of memory requested by the encoder.

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

```
 WMAEEncoderConfig *psEncConfig;
WMAEMemAllocInfoSub *mem;

/* Number of memory chunk requests by the encoder */
  nr = psEncConfig->sWMAEMemInfo.s32NumReqs;
  for(i = 0; i < nr; i++)
  {
      mem = &(psEncConfig->sWMAEMemInfo.sMemInfoSub[i]);
      if (mem->s32WMAEType == WMAE_FAST_MEMORY)
      {
          mem->app_base_ptr = malloc (mem->s32WMAESize);
          if (mem->app_base_ptr == NULL)
               /  *Error handling */
      }
      else
      {
          mem->app_base_ptr = malloc (mem->s32WMAESize);
          if (mem->app_base_ptr == NULL)
               /*Error handling */
      }
  }
```

Step 5: Initialization routine

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

C prototype:
*tWMAEncodeStatus eInitWMAEncoder(WMAEEncoderConfig *psEncodeConfig);*

Arguments:

• *psEncodeConfig*	- Pointer to encoder configuration structure.

Return value:

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

Step 6: Memory allocation for input buffer

The application has to allocate the memory needed for the input buffer. The size of the input buffer is size of one frame PCM samples, which can be calculated by:

*sizeof(short)*sEncParams.pFormat.nSamplesPerFrame*sEncParams.pFormat.nChannels*

The variables *sEncParams.pFormat.nSamplesPerFrame and sEncParams.pFormat.nChannels* are filled by the encoder in the initialization routine.

Step 7: Memory allocation for output buffer

The application has to allocate memory for the output buffers to hold the encoded WMA bit streams of one frame PCM samples. The pointer to this output buffer needs to be passed to the eWMAEncodeFrame function. The size of the output buffer is:

*sizeof(char)*sEncParams.WMAE_packet_byte_length*

The variable *sEncParams.WMAE_packet_byte_length* is a member of structure `WMAEEncoderParams,` its value is filled by the encoder in the initialization routine.

Step 8: Call the frame encoder routine

The main WMA Encoder function is *eWMAEncodeFrame*. This function encodes the PCM samples in the input buffer to generate one packet of encoder output in each call. The output buffer is filled with WMA bit stream.

C prototype:

```
tWMAEncodeStatus eWMAEncodeFrame( WMAEEncoderConfig* psEncodeConfig,

                                    WMAE_INT16 *pInputBuffer,

                                   WMAE_INT8 *pOutputBuffer,
                                   WMAE_Bool bNoMoreData );
```

Arguments:

• `psEncodeConfig`	- Encoder config parameter structure pointer
• `pInputBuffer`	- Pointer to the input buffer to hold the PCM samples
• `pOutputBuffer`	- Pointer to the output buffer to hold the encoded bit streams
• `bNoMoreData`	- Flag to show if there is enough samples in the input buffer. `bNoMoreData = 0` indicates that there are more samples to encode. `bNoMoreData = 1` indicates that there are no more samples to encode.

Return value:

• WMA_Succeeded	- Indicates encoding was successful.
• WMA_NoMoreFrames	-Indicates there are no more frames and the encoder can be finished.
• Others	- Indicates error

Example pseudo code for calling the main encode routine of the encoder

```
while( iStatus != WMA_NoMoreFrames )
{

        m_bNoMoreData = InsertNewSamples( App_pAudioInput, (unsigned
char*)wavInBuf,
                sizeof(short) *
sEncParams.pFormat.nSamplesPerFrame*sEncParams.pFormat.nChannels );

        iStatus =
eWMAEncodeFrame( psEncConfig,wavInBuf,wmaOutBuf,m_bNoMoreData);
        if((int)iStatus < 0)
        {
              fprintf(stderr,"WMA Encode Frame Failed!\n");
                  break;
        }

#ifdef ADD_ASF
  /* Asf packetize */
          rcAsf = asf_packetize (psASFParams, wmaOutBuf, asfOutBuf,
sEncParams.WMAE_isPacketReady, sEncParams.WMAE_nEncodeSamplesDone);

        if(rcAsf != cASF_NoErr)
        {
          fprintf(stderr,"Add Asf packet failed.\n");
          break;
        }
#endif
}
```

4 Example calling Routine

The below code snippets may not compile as error handling is not extensive.

```
int main( int argc, TCHAR *argv[] )
{
  //CAudioObjectEncoder *pauenc = NULL;
  WMAEEncoderConfig *psEncConfig;
  WMAEEncoderParams sEncParams;
  WavFileIO *App_pAudioInput = NULL;
                  // file ptr to our input WAV file
  tWMAEncodeStatus iStatus;
  WMAEMemAllocInfoSub *mem;
  WMAE_INT32 i, nr, j;
//  WMAFormatInfo Format = {0};
  int m_bNoMoreData= FALSE;
  int duration = 0;
  int FrameNum = 0 ;

  ASFParams *psASFParams = NULL;
#ifdef ADD_ASF
  ASFRESULTS rcAsf = cASF_NoErr;
#endif
  short * wavInBuf = NULL;
  char * wmaOutBuf = NULL;
  char * asfOutBuf = NULL;

  TCHAR App_szInputFileName[STRING_SIZE];             // input file name
  TCHAR App_szOutputFileName[STRING_SIZE];           // output file name


  start = clock();

//  sturctSizeInfo();

  fprintf(stdout,"\nWMA Encoder sample application.\n");

  // Output the WMA8 Encoder Version Info
  printf("%s \n", WMA8ECodecVersionInfo());

#ifdef DEBUG_DUMP
    fBitstream = fopen("bitstream.dat", "wt");
    fAsf = fopen("asf.dat", "wt");
    fStepSize = fopen("fStepSize.dat", "wt");
#endif
  // initialize our encode data struct
  memset((void *)&sEncParams,0,sizeof(WMAEEncoderParams));

  sEncParams.App_szInputFileName = (TCHAR *)App_szInputFileName;
  sEncParams.App_szOutputFileName = (TCHAR *)App_szOutputFileName;
  psASFParams = (ASFParams *)alloc_align(sizeof(ASFParams));
  if(psASFParams == NULL)
  {
    fprintf(stderr,"Out of Memory!\n");
    goto Cleanup;;
  }
  memset(psASFParams,0,sizeof(ASFParams));

  // process the command line
  if(FALSE == ProcessCommandLine(&sEncParams,psASFParams,argc,argv))
  {
    UsageBanner();
    FreeContentDescriptionFields();
    exit(1);
  }

  // at this point we believe we have enough information for an 
encode...start with the input file
  App_pAudioInput = wfioNew();

  if(NULL == App_pAudioInput)
    {
      fprintf(stderr,"! Cannot create WAV read/write object.\r\n");
      iStatus = (tWMAEncodeStatus) -1;
      goto Cleanup;
    }

  iStatus = 
wfioOpen(App_pAudioInput,sEncParams.App_szInputFileName,NULL,0,wfioModeRe
ad);

  if(0 != iStatus)
    {

      fprintf(stderr,"! Unable to open the requested input file.\r\n");
      iStatus = (tWMAEncodeStatus) -1;
      goto Cleanup;
    }

  pRawOutput = rfioNew();

  if(NULL == pRawOutput)
    {
      fprintf(stderr,"! Cannot create Raw write object.\r\n");
      iStatus = (tWMAEncodeStatus) -1;
      goto Cleanup;
    }

//  iStatus = rfioOpen(pRawOutput,"port.wma",rfioModeWrite);

  iStatus = 
rfioOpen(pRawOutput,sEncParams.App_szOutputFileName,rfioModeWrite);

  if(0 != iStatus)
```

```
    {

      fprintf(stderr,"! Unable to open the requested output file.\r\n");
      iStatus = (tWMAEncodeStatus) -1;
      goto Cleanup;
    }

  // retrieve our source audio information
  sEncParams.App_pAudioInput = wfioGetFormat(App_pAudioInput);

  if(NULL == sEncParams.App_pAudioInput)
    {


      fprintf(stderr,"! Unable to read WAV header information from
requested input file.\r\n");
      iStatus = (tWMAEncodeStatus) -1;
      goto Cleanup;
    }

    // only 22, 32, 44kHz are currently supported. other sampling
    // frequencies require new excitation / masking tables. see
    // contents of exct_tbl_*.h, which can be generated from
    // floating-point code.

    if( 48000 != sEncParams.App_pAudioInput->nSamplesPerSec &&
        44100 != sEncParams.App_pAudioInput->nSamplesPerSec &&
        32000 != sEncParams.App_pAudioInput->nSamplesPerSec &&
        22050 != sEncParams.App_pAudioInput->nSamplesPerSec )
    {
        fprintf( stdout, " Only 48, 44.1, 32, and 22kHz sampling rates
are supported.\n" );
        goto Cleanup;
    }

  if( 0 == sEncParams.App_iDstAudioBitRate )
sEncParams.App_iDstAudioBitRate = 70000;

  // destination sample rate / # channels / duration are the same as the
source.
  sEncParams.App_iDstAudioSampleRate = sEncParams.App_pAudioInput-
>nSamplesPerSec;
  sEncParams.App_iDstAudioChannels   = sEncParams.App_pAudioInput-
>nChannels;
  sEncParams.App_iAudioSrcLength     =
wfioGetDataLength(App_pAudioInput);

  duration = (unsigned int)(( (unsigned long
long)sEncParams.App_iAudioSrcLength * 1000 ) /
             ( sEncParams.App_pAudioInput->nSamplesPerSec
             * sEncParams.App_pAudioInput->nBlockAlign ));

  fprintf(stdout,"\r\nSource Audio Configuration\r\n\r\n");
  fprintf(stdout,"- Bit Rate        : %d kbps\r\n",( ( 
sEncParams.App_pAudioInput->nAvgBytesPerSec + 62 ) / 125 ));
  fprintf(stdout,"- Sampling Rate   : %2.1f 
kHz\r\n",(float)sEncParams.App_pAudioInput->nSamplesPerSec / 1000 );
  fprintf(stdout,"- Channels        : %d\r\n",sEncParams.App_pAudioInput-
>nChannels );
  fprintf(stdout,"- Duration        : %d ms\r\n\r\n",duration );

  psEncConfig = (WMAEEncoderConfig 
*)alloc_align(sizeof(WMAEEncoderConfig));
  if(psEncConfig == NULL)
  {
    fprintf(stderr,"Out of Memory!\n");
    goto Cleanup;
  }

  psEncConfig->psEncodeParams = &sEncParams;

  if((iStatus = eWMAEQueryMem(psEncConfig))!= WMA_Succeeded)
  {
    fprintf(stderr,"Query Memory Failed!\n");
    goto Cleanup;
  }

  /* Number of memory chunk requests by the encoder */
  nr = psEncConfig->sWMAEMemInfo.s32NumReqs;
  for(i = 0; i < nr; i++)
  {
      mem = &(psEncConfig->sWMAEMemInfo.sMemInfoSub[i]);

      if (mem->s32WMAEType == WMAE_FAST_MEMORY)
      {
          mem->app_base_ptr = alloc_align (mem->s32WMAESize);

          if (mem->app_base_ptr == NULL)
              goto Cleanup;
      }
      else
      {
          mem->app_base_ptr = alloc_align (mem->s32WMAESize);

          if (mem->app_base_ptr == NULL)
              goto Cleanup;
      }
  }

  iStatus = eInitWMAEncoder( psEncConfig );

  if(iStatus != WMA_Succeeded)
  {
    fprintf(stderr, "** Init WMA Encoder Failed!\n");
    goto Cleanup;
  }
  sEncParams.App_iDstAudioBitRate    = sEncParams.pFormat.nAvgBytesPerSec
* 8;
  sEncParams.App_iDstAudioSampleRate = sEncParams.pFormat.nSamplesPerSec;
  sEncParams.App_iDstAudioChannels   = sEncParams.pFormat.nChannels;

  fprintf(stdout,"\r\nDestination Audio Configuration\r\n\r\n");
  fprintf(stdout,"- Bit Rate        : %d kbps\r\n", (
sEncParams.App_iDstAudioBitRate + 500 )/ 1000 );
  fprintf(stdout,"- Sampling Rate   : %2.1f
kHz\r\n",(float)sEncParams.App_iDstAudioSampleRate / 1000 );
  fprintf(stdout,"- Channels        :
%d\r\n",sEncParams.App_iDstAudioChannels );
  fprintf(stdout,"- Duration        : %d ms\r\n\r\n",duration );



  wavInBuf = (short
*)alloc_align(sizeof(short)*sEncParams.pFormat.nSamplesPerFrame*sEncParam
s.pFormat.nChannels);
  if(wavInBuf == NULL)
  {
    fprintf(stderr,"Out of Memory!\n");
    goto Cleanup;
  }
#ifdef ADD_ASF
  SetAsfParams(psASFParams,sEncParams);
  rcAsf = add_asf_file_header (psASFParams);
  if(rcAsf != cASF_NoErr)
  {
    fprintf(stderr,"Add Asf file header failed.\n");
    goto Cleanup;
  }
  asfOutBuf = (char *)alloc_align(sizeof(char)*psASFParams-
>g_asf_packet_size);
  if(asfOutBuf == NULL)
  {
    fprintf(stderr,"Out of Memory!\n");
    goto Cleanup;
  }
#endif
  wmaOutBuf = (char
*)alloc_align(sizeof(char)*sEncParams.WMAE_packet_byte_length);
  if(wmaOutBuf == NULL)
  {
    fprintf(stderr,"Out of Memory!\n");
    goto Cleanup;
  }


  // all set to encode our content
  fprintf(stdout,"* Encoding content, this may take awhile...\r\n");

  cbCurr=0;
  cbLeft = (long long) sEncParams.App_iAudioSrcLength;

    while( iStatus != WMA_NoMoreFrames )
    {

        m_bNoMoreData = InsertNewSamples( App_pAudioInput, (unsigned
char*)wavInBuf,
                sizeof(short) *
sEncParams.pFormat.nSamplesPerFrame*sEncParams.pFormat.nChannels );


        iStatus = eWMAEncodeFrame(
psEncConfig,wavInBuf,wmaOutBuf,m_bNoMoreData);
        if((int)iStatus < 0)
        {
                  fprintf(stderr,"WMA Encode Frame Failed!\n");
          goto Cleanup;
        }
        FrameNum++;
        //printf("Frame: %d encoded.\n", FrameNum);


#ifdef ADD_ASF
        //rcAsf = asf_packetize(&psASFParams,outBuf);
        rcAsf = asf_packetize (psASFParams, wmaOutBuf, asfOutBuf,
sEncParams.WMAE_isPacketReady, sEncParams.WMAE_nEncodeSamplesDone);

        if(rcAsf != cASF_NoErr)
        {
          fprintf(stderr,"Add Asf packet failed.\n");
          goto Cleanup;
        }
#endif
    }

  // see if things went as planned here
  if((int)iStatus < 0)
    {
      fprintf(stderr,"! Unable to encode content.\r\n\r\n");
      goto Cleanup;
    }

#ifdef ADD_ASF
  psASFParams->nSize = sEncParams.App_iAudioSrcLength;
  psASFParams->nSR = sEncParams.App_pAudioInput->nSamplesPerSec;


  rcAsf = update_asf_file_header( psASFParams,
      sEncParams.App_iDstAudioBitRate    / 1000,
      sEncParams.App_iDstAudioSampleRate / 1000,
      sEncParams.App_iDstAudioChannels,
      sEncParams.App_pAudioInput->nBlockAlign,
      sEncParams.pFormat.nSamplesPerFrame);
```

```
   if(rcAsf != cASF_NoErr)
  {
    fprintf(stderr,"Update Asf file header failed.\n");
    goto Cleanup;
  }
#endif

  fprintf(stdout,"* Encode of content is complete.\r\n\r\n");

  finish = clock();


Cleanup:
    if(wavInBuf)
    {
        free(wavInBuf);
    }

    if(wmaOutBuf)
    {
        free(wmaOutBuf);
    }

    if(asfOutBuf)
    {
        free(asfOutBuf);
    }
#ifdef ADD_ASF
    if(psASFParams)
    {
      free(psASFParams);
    }
#endif
    for (i = 0; i < nr; i++)
    {
        if(psEncConfig->sWMAEMemInfo.sMemInfoSub[i].app_base_ptr)
        {
            free (psEncConfig->sWMAEMemInfo.sMemInfoSub[i].app_base_ptr);
        }
    }

    if(psEncConfig)
    {
        free (psEncConfig);
    }

  //if(pauenc)
  //  auencDelete (pauenc);

  // close our input file
  if(App_pAudioInput)
    wfioClose(App_pAudioInput);
  if(sEncParams.App_pAudioInput)
    free(App_pAudioInput);
  // close output file
  if(pRawOutput)
    {
      rfioClose(pRawOutput);
      free(pRawOutput);
    }

  // free the buffers allocated for content description
  FreeContentDescriptionFields();

  exit(0);
}

int InsertNewSamples(WavFileIO *App_pAudioInput, unsigned char*
pInputCurr, const int nDataWanted )
{
    const char  iSize   = 1;
    const int cbSrc   = (int)( nDataWanted < cbLeft ?
                               nDataWanted : cbLeft );
    int cbReturnedLen = 0;

    if( 0 < cbSrc )
    {
        cbReturnedLen = GetAudioData( App_pAudioInput, pInputCurr, cbCurr,
cbSrc, iSize ) * iSize;
        if( 0 > cbReturnedLen )
        {
            // this indicates that getAudioData failed.
            assert( !"GetAudioData() failed." );
            return FALSE;
        }
    }

    if( cbReturnedLen < nDataWanted )
    {
        memset( pInputCurr + cbReturnedLen,
            0, nDataWanted - cbReturnedLen );
    }

    cbCurr += nDataWanted;
    cbLeft -= nDataWanted;

    // return true if no data remains, otherwise false.

    return ( cbSrc <= 0 );
}

int GetAudioData(WavFileIO *App_pAudioInput, unsigned char
*pbAudioDataBuffer, const long long iByteOffset,
             const int iNumBytesWanted, const char iSize )
{
  wfioSeek( App_pAudioInput, (int)iByteOffset, 0 );
  return wfioRead( App_pAudioInput, pbAudioDataBuffer,
                    iNumBytesWanted, iSize );
}
```