08-7155-API-ZCH66
OCTOBER 14, 2008
4.0

Application Programmers Interface for NB-AMR Decoder and Encoder

ABSTRACT:

Application Programmers Interface for NB AMR Decoder and Encoder

KEYWORDS:

Multimedia codecs, speech, NB AMR, Narrow Band AMR

APPROVED:

Shang Shidong

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	23-Mar-2004	Ashok Kumar	Initial Draft
0.2	26-Mar-2004	Ashok Kumar	Added review comments from Satish and Sanjay
1.0	10-Jul-2004	Ashok Kumar	Added support for MMS compliance file format
1.1	13-Sep-2004	Ashok Kumar	Updated naming convention as per code (Hungarian notation)
1.2	15-Sep-2004	Ashok Kumar	Update reference section
1.3	16-Sep-2004	Ashok Kumar	Added generic review comments of PCS team
2.0	17-Jan-2005	Ashok Kumar	Updated as per latest API
3.0	06-Feb-2006	Lauren Post	Using new format
4.0	13-Oct-08	Tao Jun	Updated doc ID, added version API

Table of Contents

Introduction

1.1 Purpose

This document gives the details of the application programmer's interface (API) of Narrow Band Adaptive Multi-Rate (NB-AMR) codec. The NB-AMR encoder compresses linear PCM speech input data at 8 kHz sampling rate to one of eight data rate modes- 12.2, 10.2, 7.9, 7.4, 6.7, 5.9, 5.15 and 4.75 kbps. NB-AMR coding scheme is based on the principle of Algebraic Code Excited Linear Prediction algorithm.

The NB-AMR algorithms also implements silence compression techniques to reduce the transmitted bit rate during the silent intervals of speech. Voice activity detector (VAD) and Comfort noise generation (CNG) algorithms are used to enable the transmission of silence descriptor (SID) frames during the periods of silence.

The NB-AMR codec is OS independent and do not assume any underlying drivers.

1.2 Scope

This document describes only the functional interface of the NB-AMR codec. It does not describe the internal design of the codec. Specifically, it describes only those functions which are required for this codec to be integrated in a system.

1.3 Audience Description

The reader is expected to have basic understanding of Speech Signal processing and NB-AMR vocoder. The intended audience for this document is the development community who wish to use the NB-AMR codec in their systems.

1.4 References

1.4.1 Standards

- **ETSI EN 301 703 V7.0.2 (1999-12)** - Digital cellular telecommunications system (Phase 2+) (GSM); Adaptive Multi-Rate (AMR); Speech processing functions; General description (GSM 06.71 version 7.0.2 Release 1998)
- **ETSI EN 301 704 V7.2.1 (2000-04)** - Digital cellular telecommunications system (Phase 2+) (GSM); Adaptive Multi-Rate (AMR) speech transcoding (GSM 06.90 version 7.2.1 Release 1998).
- **ETSI EN 301 705 V7.1.1 (2000-04)** - Digital cellular telecommunications system (Phase 2+); Substitution and muting of lost frames for Adaptive Multi Rate (AMR) speech traffic channels (GSM 06.91 version 7.1.1 Release 1998).

- **ETSI EN 301 706 V7.1.1 (1999-12)** - Digital cellular telecommunication system (Phase 2+); Comfort noise aspects for Adaptive Multi-Rate (AMR) speech traffic channels (GSM 06.92 version 7.1.1 Release 1998)
- **ETSI EN 301 707 V7.3.1 (2001-03)** - Digital cellular telecommunications system (Phase 2+); Discontinuous Transmission (DTX) for Adaptive Multi-Rate (AMR) speech traffic channels (GSM 06.93 version 7.3.1 Release 1998).
- **ETSI EN 301 708 V7.1.1 (1999-12)** - Digital cellular telecommunications system (Phase 2+); Voice Activity Detector (VAD) for Adaptive Multi-Rate (AMR) speech traffic channels; General description (GSM 06.94 version 7.1.1 Release 1998).
- **ETSI EN 301 712 V7.4.1 (2000-09)** - Digital cellular telecommunications system (Phase 2+); Adaptive Multi Rate (AMR) speech; ANSI-C code for the AMR speech codec (GSM 06.73 version 7.4.1 Release 1998).
- **ETSI EN 301 713 V7.0.3 (2000-10)** - Digital cellular telecommunications system (Phase 2+); Test sequences for the Adaptive Multi-Rate (AMR) speech codec (GSM 06.74 version 7.0.3 Release 1998).
- **ITU-T Recommendation G.711 (1988)** – Coding of analogue signals by pulse code modulation Pulse code modulation (PCM) of voice frequencies.
- **3GPP TS 26.101 V5.0.0 (2002-06) -** Adaptive Multi-Rate (AMR) speech codec frame structure.

1.4.2 General references

- E. Paksoy, J.C.D Martin, Alan McCree, C.G.Gerlach, Anand Anandakumar, Wai-Ming Lai and Vishu Viswanathan, ICASS Proceeding 1999 “An Adaptive Multi-Rate Speech Coder for Digital Cellular Telephony”
- Real-Time Transport Protocol (RTP) Payload Format and File Storage Format for the Adaptive Multi-Rate (AMR) and Adaptive Multi-Rate Wideband (AMR-WB) Audio Codecs “RFC3267”

1.4.3 Freescale Multimedia References

- NB AMR Codec Application Programming Interface – nbamr_codec_api.doc
- NB AMR Codec Requirements Book – nbamr_codec_reqb.doc
- NB AMR Codec Test Plan - nbamr_codec_test_plan.doc
- NB AMR Codec Release notes - nbamr_codec_release_notes.doc
- NB AMR Codec Test Results – nbamr_codec_test_results.doc
- NB AMR Codec Performance Results – nbamr_codec_perf_results.doc
- NB AMR Interface Common Header – nbamr_common_api.h
- NB AMR Interface Decoder Header – nbamr_dec_api.h
- NB AMR Interface Encoder Header – nbamr_enc_api.h
- NB AMR Decoder Application Code – nbamr_dectest.c
- NB AMR Encoder Application Code – nbamr_enctest.c

1.5 Definitions, Acronyms, and Abbreviations

TERM/ACRONYM	DEFINITION
ACELP	Algebraic Code Excited Linear Prediction
API	Application Programming Interface
ARM	Advanced RISC Machine
CNG	Comfort Noise Generation
DTX	Discontinuous Transmission
ETSI	European Standard Telecommunications Series
FSL	Freescale
ITU	International Telecommunication Union
LSP	Line Spectral Pair
LP	Linear Prediction
MIPS	Million Instructions per Second
NB AMR	Narrow and Adaptive Multi-Rate Codec
OS	Operating System
PCM	Pulse Code Modulation
RVDS	ARM RealView Development Suite
SCR	Source Controlled Rate
SID	Silence Insertion Descriptor
TBD	To Be Determined
UNIX	Linux PC x/86 C-reference binaries
VAD	Voice Activity Detection

1.6 Document Location

docs/nb_amr

2 API Description

This section describes the steps followed by the application to call the NB-AMR encoder and decoder. During each step the data structures and the functions used will be explained. Pseudo code is given at the end of each step.

2.1 Encoder API Data Types

The member variables inside the structure are prefixed as AMRE or APPE together with data types prefix to indicate if that member variable needs to be initialized by the encoder or application calling the encoder.

Step 1: Print version information

The application can get version information such as version number and build time by calling the below function.

C prototype:

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

Arguments:

- None.

Return Value:

- Returns a sting which includes version information

Step 2: Allocate memory for Encoder config parameter structure

The application allocates memory for below mentioned structure.

```
/* encoder parameter structure */
typedef struct
{
      sAMREMemAllocInfoType   sAMREMemInfo;
      NBAMR_Void              *pvAMREEncodeInfoPtr;
      NBAMR_U8                *pu8APPEInitializedDataStart;
      NBAMR_S16               s16APPEDtxFlag;
      NBAMR_S8                **pps8APPEModeStr;
      NBAMR_S8                **pps8AMREUsedModeStr;
      NBAMR_U32               *pu32AMREPackedSize;
            /* size of packed data in case of MMS, IF1 and IF2 */
      NBAMR_U8                u8BitStreamFormat;
      NBAMR_U8                u8NumFrameToEncode;
            /* number of frame to be encoded */
} sAMREEncoderConfigType;
```

Description of the encoder parameter structure *sAMREEncoderConfigType*

sAMREMemInfo

This is memory information structure. The application needs to call the function eAMREQueryMem to get the memory requirements from encoder. The encoder will fill this structure with its memory requirements. This will be discussed in **step 3**.

pvAMREEncodeInfoPtr

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

pu8APPEInitializedDataStart

The application has to assign this pointer with the symbol supplied in the header file. This symbol is the start address of the initialized data that the encoder uses. The encoder needs to know this as the OS can relocate the data tables of the NB-AMR encoder every time the application is invoked.

s16APPEDtxFlag

The application shall set the value of this variable to 0 (disable DTX) or 1 (enable DTX). Encoder reads this value during its initialization.

pps8APPEModeStr

This is pointer to pointer to mode string. Application should set this value before calling encode frame routine. Double pointer is required to process multiple frame in single call of encode frame.

pps8AMREUsedModeStr

This is pointer to pointer to the mode used by encoder. Encoder will set this to a string corresponding to requested encoding mode or MRDTX in absence of valid speech. Double pointer is needed in case numframe is set to more than one.

pu32AMREPackedSize

Encoder will return size (in bytes) of packed data through this variable. Application can use this information for writing packed data to a file or buffer. In case of IF1 format (non octet aligned), coded data will be returned in number of bits.

u8BitStreamFormat

Application need to set this to required format in which coded data should be given to application. Default value of this is ETSI. This needs to be configured by application.

u8NumFrameToEncode

This should be set to appropriate value before calling encode frame function. Value of this should be between 1 and 255. This needs to be configured by application.

Example pseudo code for this step:

```
/* allocate memory for the encoder parameter */
sAMREEncoderConfigType *psEncConfig;

/* allocate fast memory for encoder config */
psEncConfig = (sAMREEncoderConfigType*)\
                  alloc_fast (sizeof(sAMREEncoderConfigType);


/* Fill up the relocated data position (not used) */
psEncConfig->pu8APPEInitializedDataStart = NBAMR_BEGIN_DATA;


/* allocate memory for encoder to use */
```



```
psEncConfig->pvAMREEncodeInfoPtr = NULL;
/* enable DTX */
psEncConfig->s16DtxFlag = 1;
psEncConfig->u8BitStreamFormat = NBAMR_ETSI;
psEncConfig->u8NumFrameToEncode = 1;
```

Step 3: Get the encoder memory requirements

The NB-AMR encoder does not do any dynamic memory allocation. The application calls the function *eAMREQueryMem* to get the encoder memory requirements. This function must be called before any other encoder functions are invoked.

The function prototype of *eAMREQueryMem* is:

C prototype:

```
eAMREReturnType eAMREQueryMem (sAMREEncoderConfigType  *psEncConfig);
```

Arguments:

- psEncConfig - Encoder config pointer.

Return value:

- E_NBAMRE_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 */
        NBAMR_S32                       s32AMRENumMemReqs;
        sAMREMemAllocInfoSubType
                        asMemInfoSub[NBAMR_MAX_NUM_MEM_REQS];
} sAMREMemAllocInfoType;
```

Description of the structure *sAMREMemAllocInfoType*

s32AMRENumMemReqs

The number of memory chunks requested by the encoder.

asMemInfoSub

This structure contains each chunk's memory config parameters.

```
typedef struct
 {
      NBAMR_S32         s32AMRESize;            /* Size in bytes */
      NBAMR_S32         s32AMREMemType;/* Memory is static or scratch */
      NBAMR_S32         s32AMREMemTypeFs; /* Memory type FAST or SLOW */
      NBAMR_U8          u8AMREMemPriority;   /* Priority level */
      NBAMR_Void        *pvAPPEBasePtr;
            /* Pointer to the base memory, that will be allocated and
            filled by the application*/
} sAMREMemAllocInfoSubType;
```

Description of the structure *sAMREMemAllocInfoSubType*

s32AMRESize

The size of each chunk in bytes.

s32AMREMemType

The memory description field indicates whether requested chunk of memory is static or scratch. Codec will update this flag to STATIC or SCRATCH based on whether the requested memory chunk is used as STATIC or as SCRATCH memory. It will depend on the application to make use of this information.

s32AMREMemTypeFs

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 NBAMR_SLOW_MEMORY or external memory, NBAMR_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 requests for a NBAMR_FAST_MEMORY for which the application allocates a NBAMR_SLOW_MEMORY, the encoder will still encode, but the performance (MHz) will suffer.)

u8AMREMemPriority

This indicates the priority level of the memory type. The type of memory can be NBAMR_SLOW_MEMORY or external memory, NBAMR_FAST_MEMORY or internal memory. In case the type of memory is NBAMR_FAST_MEMORY then the field *u8AMREMemPriority* indicates the importance or the priority of the request. A priority value of zero indicates highest priority and 255 indicates lowest priority.

pvAPPEBasePtr

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 *pvAPPEBasePtr*. 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

```
eAMREReturnType eRetVal;

/* Query for memory */
eRetVal = eAMREQueryMem (psEncConfig);

if (eRetVal != E_NBAMRE_OK)
      return NBAMR_FAILURE;
```

Step 4: Allocate Data Memory for the encoder

In this step, the application allocates the memory as required by the NB-AMR encoder and fills up the base memory pointer *'pvAPPEBasePtr' of* '*sAMREMemAllocInfoSubType*' 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 is given below.

```
sAMREMemAllocInfoSubType *psMem;
```

```
/* define local variable if static/scratch type information
   needed for chunks */
NBAMR_S32 s32MemDetail[NBAMR_MAX_NUM_REQS];

/* Number of memory chunks requested by the encoder */
nr = psEncConfig->sAMREMemInfo.s32AMRENumMemReqs;

for(i = 0; i < nr; i++)
{
      psMem = &(psEncConfig-> sAMREMemInfo.asMemInfoSub[i]);
      /* Get whether mem is for STACK or STATIC */
      s32MemDetail[i] = psMem-> s32AMREMemType;
      if (psMem->s32AMREMemTypeFs == NBAMR_FAST_MEMORY)
      {
            /* If application does not have enough memory to allocate
            in fast memory, it can check priorty
            of requested chunk (psMem-> u8AMREMemPriority) and
            allocate accordingly */

            /* This function allocates memory in internal memory */
            psMem->pvAPPEBasePtr = alloc_fast(psMem->s32AMRESize);
      }
      else
      {
            This function allocates memory in external memory */
            psMem->pvAPPEBasePtr = alloc_slow(psMem->s32AMRESize);
      }
}
```

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

Step 5: Initialization routine

All initializations required for the encoder are done in eAMREEncodeInit. This function must be called before the main encode frame function is called.

C prototype:

```
eAMREReturnType eAMREEncodeInit (sAMREEncoderConfigType *);
```

Arguments:

- Pointer to encoder configuration structure

Return value:

- E_NBAMRE_OK - Initialization successful.
- Other codes - Initialization Error

Example pseudo code for calling the initialization routine of the decoder

```
/* Initialize the NB-AMR encoder. */
eRetVal = eAMREEncodeInit (psEncConfig);

if (eRetVal != E_NBAMRE_OK)
```

```
        return NBAMR_FAILURE;
```

Step 6: Memory allocation for input buffer

The application has to allocate (aligned to 2-byte boundary) the memory needed for the input buffer. It is desirable to have the input buffer allocated in NBAMR_FAST_MEMORY, as this may improve the performance (MHz) of the encoder. The size of input buffer should be N (number of frame to be encoded) times size of speech frame size i.e. N*160 words. Pointer to the input buffer needs to be passed to encode frame routine.

Example pseudo code for allocating the input buffer

```
/* allocate memory for input buffer */
ps16InBuf = alloc_fast(u8NumFrameToEncode*L_FRAME*sizeof (NBAMR_S16));
```

Special Consideration

13-bit uniform PCM input sample should be left aligned in 16 bit word boundary with remaining 3 bits set to zero as shown below.

MSB	LSB	zeroes

Step 7: Memory allocation for output buffer

The application has to allocate memory for the output buffers to hold the encoded bitstream corresponding to one frame of speech sample at maximum supported bitrate, i.e.12.2 kbps. Based on output file format it can be 250 words (for ETSI format) or maximum of 34 bytes (corresponding to IF1 format, MMS and IF2 needs maximum of 32 bytes). The pointer to this output buffer needs to be passed to the eAMREEncodeFrame 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 encoder marginally.

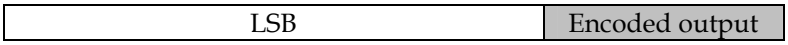
Example pseudo code for allocating memory for output buffer

```
If (u8BitStreamFormat == NBAMR_ETSI)
{
      ps16OutBuf=alloc_slow(u8NumFrameToEncode*SERIAL_FRAMESIZE*
                  sizeof (NBAMR_S16));
}
else
{
      ps16OutBuf=alloc_slow(u8NumFrameToEncode*NBAMR_MAX_PACKED_SIZE*
                  sizeof (NBAMR_S16));
}
```


Note: NBAMR_MAX_PACKED_SIZE is set to 34 bytes and defined in nb_amr_common_api.h file.

Special Consideration

In case of ETSI format encoder output will be in one bit per byte format and that bit will be placed at least significant bit position as shown below.



In Non ETSI format output will be packed as per standard.

Step 8: Call the frame encode routine

The main NB-AMR encoder function is eAMREEncodeFrame. This function encodes the 13-bit uniform PCM sample and writes bitstream to output buffer.

C prototype:

```
eAMREReturnType        eAMREEncodeFrame (
                              sAMREEncoderConfigType *psEncConfig,
                              NBAMR_S16 *ps16InBuf,
                              NBAMR_S16 *ps16OutBuf);
```

Arguments:

- psEncConfig - Pointer to encoder config structure
- ps16InBuf - Pointer to input speech buffer
- ps16OutBuf - Pointer to output (encoded) buffer

Return value:

- *E_NBAMRE_OK* - Indicates encoding was successful.
- **Others** **- Indicates error**

Example pseudo codes for calling the main encode routine of the encoder.

```
while (NBAMRE_TRUE)
{
      allocate memory for pps8APPEModeStr based on number of frame to
      be encoded;
      psEncConfig->pps8APPEModeStr = application requested mode;

      allocate memory for pps8AMREUsedModeStr based on number of frame
      to be encoded;

      eRetVal=eAMREEncodeFrame(psEncConfig, ps16InBuf, ps16OutBuf);
      if (eRetVal  != E_NBAMRE_OK)
            return NBAMR_FAILURE;
}
```

Step 9: Free memory

The application should release all the memory it allocated before exiting the encoder.

```
free (ps16OutBuf);
free (ps16InBuf);
for (i=0; i<nr; i++)
{
      free (psEncConfig->sAMREMemInfo.asMemInfoSub[i].pvAPPEBasePtr);
}
free (psEncConfig);
```

2.2 Decoder API Data Types

The member variables inside the structure are prefixed as AMRD or APPD together with data types prefix to indicate if that member variable needs to be initialized by the decoder or application calling the decoder.

Step 1: Print version information

The application can get version information such as version number and build time by calling the below function.

C prototype:

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

Arguments:

- None.

Return Value:

- Returns a sting which includes version information

Step 2: Allocate memory for Decoder config parameter structure

The application allocates memory for below mentioned structure.

/* decoder parameter structure */

```
typedef struct
{
    sAMRMemAllocInfoType  sAMRDMemInfo;
    NBAMR_Void           *pvAMRDDecodeInfoPtr;
    NBAMR_U8             *pu8APPDInitializedDataStart;
    NBAMR_U8              u8RXFrameType;
            /* whether frame is contains RX frame */
    NBAMR_U8              u8BitStreamFormat;   /* Requested file format */
    NBAMR_U8              u8NumFrameToDecode;
            /* number of frame to be decoded */
}sAMRDDecoderConfigType;
```

Description of the decoder parameter structure *sAMRDDecoderConfigType*

sAMRDMemInfo

This is a memory information structure. The application needs to call the function eAMRDQueryMem to get the memory requirements from decoder. The decoder will fill this structure with its memory requirements. This will be discussed in **step 3**.

pvAMRDDecodeInfoPtr

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.

pu8APPDInitializedDataStart

The application has to assign this pointer with the symbol supplied in the header file. This symbol is the start address of the initialized data that the decoder uses. The decoder needs to know this as the OS can relocate the data tables of the NB-AMR decoder every time the application is invoked.

u8RXFrameType

Flag to indicate whether frame to be decoded contains RX frame type. This needs to be configured by application. Default is TX frame type.

u8BitStreamFormat

This needs to be configured by application before calling decode frame function. Default value of this is NBAMR_ETSI.

u8NumFrameToDecode

This needs to be configured by application based on how many frames need to be decoded when eAMRDDecodeFrame function is called. The value of this should be between 1 and 255.

Example pseudo code for this step:

```
/* allocate memory for the decoder parameter */
sAMRDDecoderConfigType *psDecConfig;
psDecConfig = (sAMRDDecoderConfigType *)
                              alloc (sizeof(sAMRDDecoderConfigType);
/* Fill up the Relocated data position (not used) */
psDecConfig->pu8APPDInitializedDataStart = BEGIN_AMRD_DATA;

/* Initialize to NULL */
psDecConfig->pvAMRDDecodeInfoPtr = NULL;

psDecConfig->u8RXFrameType = 0;     /* frame does not contain RX frame */
psDecConfig->u8BitStreamFormat = NBAMR_ETSI;
psDecConfig->u8NumFrameToDecode = 1;
```

Step 3: Get the decoder memory requirements

The NB-AMR decoder does not do any dynamic memory allocation. The application calls the function *eAMRDQueryMem* to get the decoder memory requirements. This function must be called before any other decoder functions are invoked.

The function prototype of *eAMRDQueryMem* is:

C prototype:

```
eAMRDReturnType eAMRDQueryMem (sAMRDDecoderConfigType * psDecConfig);
```

Arguments:

- psDecConfig - Decoder configuration pointer.

Return value:

- E_NBAMRD_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 */
      NBAMR_S32               s32AMRDNumMemReqs;
      sMRDMemAllocInfoSubType
                        asMemInfoSub[NBAMR_MAX_NUM_MEM_REQS];
} sAMRMemAllocInfoType;
```

Description of the structure *sAMRMemAllocInfoType*

s32AMRDNumMemReqs

The number of memory chunks requested by the decoder.

asMemInfoSub

This structure contains each chunk's memory configuration parameters.

```
typedef struct
{
      NBAMR_S32   s32AMRDSize;      /* Size in bytes */
      NBAMR_S32   s32AMRDMemType; /*Memory is STATIC or SCRATCH */
      NBAMR_S32   s32AMRDMemTypeFs; /* Memory type FAST or SLOW */
      NBAMR_U8    u8AMRDMemPriority;      /* Priority level */
      NBAMR_Void  *pvAPPDBasePtr;
            /* Pointer to the base memory, which will be allocated
             and filled by the application*/
} sAMRDMemAllocInfoSubType
```

Description of the structure sAMRDMemAllocInfoSubType

s32AMRDSize

The size of each chunk in bytes.

s32AMRDMemType

The memory description field indicates whether requested chunk of memory is static or scratch. Codec will update this flag to STATIC or SCRATCH based on whether the requested memory chunk is used as STATIC or as SCRATCH memory. It will depend on application to make use of this information.

s32AMRDMemTypeFs

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 NBAMR_SLOW_MEMORY or external memory, NBAMR_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 NBAMR_FAST_MEMORY for which the application allocates a NBAMR_SLOW_MEMORY, the decoder will still decode, but the performance (MHz) will suffer.)

u8AMRDMemPriority

This indicates the priority level of the memory type. The type of memory can be NBAMR_SLOW_MEMORY or external memory, NBAMR_FAST_MEMORY or internal memory. In case the type of memory is NBAMR_FAST_MEMORY then the field *u8AMRDMemPriority* indicates the importance or the priority of the request. A priority value of zero indicates highest priority and 255 indicates lowest priority.

pvAPPDBasePtr

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 *pvAPPDBasePtr*. 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

```
eAMRDReturnType eRetVal;

/* Query for memory */
eRetVal = eAMRDQueryMem (psDecConfig);

if (eRetVal != E_NBAMRD_OK)
      return NBAMR_FAILURE;
```

Step 4: Allocate Data Memory for the decoder

In this step, the application allocates the memory as required by the NB-AMR decoder and fills up the base memory pointer '*pvAPPDBasePtr' of 'sAMRDMemAllocInfoSubType*' 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 is given below.

```
sAMRDMemAllocInfoSubType *psMem;
/* to get static/stack mem detail */
NBAMR_S32 s32MemDetail[NBAMR_MAX_NUM_MEM_REQS];

/* Number of memory chunks requested by the encoder */
nr = psDecConfig->sAMRDMemInfo.s32AMRDNumMemReqs;

for(i = 0; i < nr; i++)
{
      psMem = &(psDecConfig-> sAMRDMemInfo.asMemInfoSub[i]);

      /* Get detail of static/scratch memory */
      s32MemDetail = psMem-> s32AMRDMemType:

      if (psMem->s32AMRDMemTypeFs == NBAMR_FAST_MEMORY)
      {
            /* If application does not have enough memory to allocate
            in fast memory, it can check priorty
            of requested chunk (psMem-> u8AMRDMemPriority) and
            allocate accordingly */
```



```
            /* this function allocates memory in internal memory */
            psMem->pvAPPDBasePtr = alloc_fast(psMem->s32AMRDSize);
      }
      else
      {
            This function allocates memory in external memory */
            psMem->pvAPPDBasePtr = alloc_slow(psMem->s32AMRDSize);
      }
}
```

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

Step 5: Initialization routine

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

C prototype:

```
eAMRDReturnType eAMRDDecodeInit (sAMRDDecoderConfigType *);
```

Arguments:

- Pointer to decoder configuration structure

Return value:

- E_NBAMRD_OK - Initialization successful.
- Other codes - Initialization Error

Example pseudo code for calling the initialization routine of the decoder

```
/* Initialize the NB-AMR decoder. */
eRetVal = eAMRDDecodeInit (psDecConfig);
if (eRetVal != E_NBAMRD_OK)
      return NBAMR_FAILURE;
```

Step 6: Memory allocation for input buffer

The application has to allocate (2 byte aligned) the memory needed for the input buffer. Based on input file format it can be 250 words (for ETSI format) or 34 bytes (corresponding to IF1 format, MMS and IF2 needs 32 bytes only). It is desirable to have the input buffer allocated in NBAMR_FAST_MEMORY, as this may improve the performance (MHz) of the decoder. Pointer to the input buffer needs to be passed to decode frame routine.

Example pseudo code for allocating the input buffer

```
if(u8BitStreamFormat == NBAMR_ETSI)
{
      /* allocate SERIAL_FRAME_SIZE = 250 words */
      in_buf = alloc_fast (u8NumFrameToDecode*SERIAL_FRAMESIZE*sizeof
                  (NBAMR_S16));
```

```
}
else
{
      /*output format is MMS compliance
      /* allocate NBAMR_MAX_PACKED_SIZE = 32 bytes */
      in_buf=alloc_fast (u8NumFrameToDecode*
            (NBAMR_MAX_PACKED_SIZE/2)* sizeof(NBAMR_S16));
}
```

Step 7: Memory allocation for output buffer

The application has to allocate memory for the output buffers to hold the decoded PCM sample corresponding to one speech frame (160 words). The pointer to this output buffer needs to be passed to the eAMRDDecodeFrame 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,
 L_FRAME=160 as defined in Appendix */
ps16OutBuf=alloc_slow(u8NumFrameToDecode *
                        L_FRAME * sizeof  (NBAMR_S16));
```

Special Consideration

Output of decoder is 13-bit uniform PCM sample left aligned in 16 bit word boundary with remaining 3 least significant bits set to zero as shown below.

MSB	LSB	zeroes

Step 8: Call the frame decode routine

The main NB-AMR decoder function is eAMRDDecodeFrame. This function decodes the NB-AMR bitstream and writes bitstream to PCM play out buffer.

C prototype:

```
eAMRDReturnType         eAMRDDecodeFrame (
                              sAMRDDecoderConfigType *psDecConfig,
                              NBAMR_S16 *ps16InBuf,
                              NBAMR_S16 *ps16OutBuf);
```

Arguments:

- psDecConfig - Pointer to decoder configuration structure
- ps16InBuf - Pointer to NB-AMR bitstream buffer
- ps16OutBuf - Pointer to output (decoded) buffer

Return value:

- E_NBAMRD_OK - Indicates decoding was successful.
- **Others** **- Indicates error**

Example pseudo codes for calling the main decode routine of the decoder.

```
while (NBAMRD_TRUE)
{
      eRetVal=eAMRDDecodeFrame(psDecConfig,  ps16InBuf, ps16OutBuf);
      if (eRetVal  != E_NBAMRD_OK)
            return NBAMR_FAILURE;
}
```

Step 9: Free memory

The application should release memory before exiting the decoder.

```
free (ps16OutBuf);
free (ps16InBuf);
for (i=0; i<nr; i++)
{
      free (psDecConfig->sAMRDMemInfo.asMemInfoSub[i].pvAPPDBasePtr);
}
free (psDecConfig);
```

3 Example calling Routine

3.1 Example calling routine for NB-AMR Encoder

Below example code gives a guideline for calling NB-AMR encoder.

```
/*****************************************************************
*                          INCLUDE FILES
*****************************************************************/
#include <stdio.h>  /* for declaration of FILE */
#include <string.h>
#include <stdlib.h>
#include "nb_amr_enc_api.h"
#include "nbamr_enctest.h"
/*****************************************************************/

int main (NBAMR_S32 s32Argc, NBAMR_S8 *ps8Argv[])
{
    NBAMR_S16  *ps16InBuf = NULL;    /*pointer to input speech
                                     samples*/
    NBAMR_S16  *ps16OutBuf = NULL;  /* pointer to output (encoded)
                                     buffer */
    NBAMR_S8   *ps8ModeStr = NULL;
    sAMREMemAllocInfoSubType *psMem;  /*pointer to encoder sub-memory
                                        */
    sAMREEncoderConfigType   *psEncConfig;  /* Pointer to encoder
                                               config  structure */
    eAMREReturnType  eRetVal;              /* local variable */
    NBAMR_S16  s16NumMemReqs;
    NBAMR_S16  s16Counter;
    NBAMR_S16 s16DtxFlag=0;       /* DTX enable/disable flag */
    NBAMR_S8 s8UseModeFile=0;
    NBAMR_S32 s32Frame;
    FILE *pfFileSpeech = NULL;       /* File of speech data  */
    FILE *pfFileSerial = NULL;       /* File of coded bits   */
    FILE *pfFileModes = NULL;        /* File with mode information */

    NBAMR_U8 u8BitStreamFormat = NBAMR_ETSI;    /* default format */

    NBAMR_U8 u8NumFrame = 1;    /* number of speech frame to be encoded
                                   in single call of EncodeFrame */
    NBAMR_S16 s16Size = 0;

   /* Get the version info of the sign-on NB AMR */
   fprintf(stderr,"Running %s\n",eAMREVersionInfo());

    eRetVal = eAMREProcessCmdLineOptions (s32Argc, &ps8Argv[0],
                  &s16DtxFlag, &ps8ModeStr, &s8UseModeFile,
                  &u8BitStreamFormat, &u8NumFrame,
                  &pfFileSpeech, &pfFileSerial, &pfFileModes);
    if (eRetVal != E_NBAMRE_OK)
    {
```

```
    exit(1);
}

/* allocate fast memory for encoder config structure */
psEncConfig = (sAMREEncoderConfigType *)\
                        alloc_fast(sizeof(sAMREEncoderConfigType));
if (psEncConfig == NULL)
{
    return NBAMR_FAILURE;
}

/* allocate memory for encoder to use */
psEncConfig->pvAMREEncodeInfoPtr = NULL;
/* Not used */
psEncConfig->pu8APPEInitializedDataStart = NULL;

/* Set DTX flag */
psEncConfig->s16APPEDtxFlag = s16DtxFlag;

psEncConfig->u8BitStreamFormat = u8BitStreamFormat;
psEncConfig->u8NumFrameToEncode= u8NumFrame;    /* number of frame
                                                   to be encoded */
/* encoded data size */
psEncConfig->pu32AMREPackedSize = (NBAMR_U32 *)
                    alloc_fast(u8NumFrame*sizeof(NBAMR_U32));

/* user requested mode */
psEncConfig->pps8APPEModeStr = alloc_fast(u8NumFrame *
                                sizeof(NBAMR_S8 *));
/* used mode by encoder */
psEncConfig->pps8AMREUsedModeStr = alloc_fast(u8NumFrame *
                                sizeof(NBAMR_S8*));
for (s16Counter =0; s16Counter<u8NumFrame; s16Counter++)
{
    /* set user requested encoding mode */
    psEncConfig->pps8APPEModeStr[s16Counter] = ps8ModeStr;
}

for (s16Counter =0; s16Counter<u8NumFrame; s16Counter++)
{
    /* allocate memory to strore used mode, used mode value will be
        updated by encoder libarary
     */
    psEncConfig->pps8AMREUsedModeStr[s16Counter] =
                            alloc_fast(6*sizeof(NBAMR_S8*));
}

/* initialize packed data variable for all the frames */
for (s16Counter=0; s16Counter<u8NumFrame; s16Counter++)
{
    *(psEncConfig->pu32AMREPackedSize+s16Counter) = 0;
}

/* initialize config structure memory to NULL */
```

```
for(s16Counter = 0; s16Counter <NBAMR_MAX_NUM_MEM_REQS;
  s16Counter++)
{
    (psEncConfig-
  >sAMREMemInfo.asMemInfoSub[s16Counter].pvAPPEBasePtr) = NULL;
}

/* Find encoder memory requiremet */
eRetVal = eAMREQueryMem (psEncConfig);

if (eRetVal != E_NBAMRE_OK)
{
    /* de-allocate memory allocated for encoder config */
    mem_free(psEncConfig);
    return NBAMR_FAILURE;
}
/* Number of memory chunk requested by the encoder */
  s16NumMemReqs = psEncConfig->sAMREMemInfo.s32AMRENumMemReqs;
/* allocate memory requested by the encoder*/
  for(s16Counter = 0; s16Counter <s16NumMemReqs; s16Counter++)
{
    psMem = &(psEncConfig->sAMREMemInfo.asMemInfoSub[s16Counter]);
    if (psMem->s32AMREMemTypeFs == NBAMR_NBAMR_FAST_MEMORY)
    {
        psMem->pvAPPEBasePtr = alloc_fast(psMem->s32AMRESize);
        if (psMem->pvAPPEBasePtr == NULL)
        {
            mem_free(psEncConfig);
            return NBAMR_FAILURE;
        }
    }
    else
    {
        psMem->pvAPPEBasePtr = alloc_slow(psMem->s32AMRESize);
        if (psMem->pvAPPEBasePtr == NULL)
        {
            mem_free(psEncConfig);
            return NBAMR_FAILURE;
        }
    }
}
  /* Call encoder init routine */
  eRetVal = eAMREEncodeInit (psEncConfig);
  if (eRetVal != E_NBAMRE_OK)
  {
        /* free all the dynamic memory and exit */
    eRetVal=eAMREEncodeExit(psEncConfig, s8UseModeFile, ps16InBuf,
                                ps16OutBuf);
    return NBAMR_FAILURE;
  }

  /* allocate memory for input buffer */
  ps16InBuf = alloc_fast((psEncConfig->u8NumFrameToEncode *
                          L_FRAME) * sizeof (NBAMR_S16));
```

```
if (ps16InBuf == NULL)
{
        /* free all the dynamic memory and exit */
    eRetVal=eAMREEncodeExit(psEncConfig, s8UseModeFile, ps16InBuf,
                            ps16OutBuf);
    return NBAMR_FAILURE;
}
/* allocate memory for output buffer (unpacked) */
if (psEncConfig->u8BitStreamFormat == NBAMR_ETSI)
{
    ps16OutBuf = alloc_fast ((psEncConfig-
  >u8NumFrameToEncode*SERIAL_FRAMESIZE)*
                          sizeof (NBAMR_S16));
}
else  /* it is either, mms or if1 or if 2 format */
{
    ps16OutBuf = alloc_fast ((psEncConfig->u8NumFrameToEncode*
            ((NBAMR_MAX_PACKED_SIZE/2)+(NBAMR_MAX_PACKED_SIZE%2)))*
                            sizeof (NBAMR_S16));
}
if (ps16OutBuf == NULL)
{
        /* free all the dynamic memory and exit */
    eRetVal=eAMREEncodeExit(psEncConfig, s8UseModeFile, ps16InBuf,
                            ps16OutBuf);
    return NBAMR_FAILURE;
}
if (psEncConfig->u8BitStreamFormat == NBAMR_MMSIO)
{
   /* write AMR magic number to indicate single channel AMR file
        format */
    fwrite(NBAMR_MAGIC_NUMBER, sizeof(NBAMR_U8),
            strlen(NBAMR_MAGIC_NUMBER), pfFileSerial);
    fflush(pfFileSerial);
}
  /*************************************************************
   * Encode speech frame till end of frame
   *************************************************************/
s32Frame = 0;
while ((s16Size=s16AMREReadSpeechSample (pfFileSpeech, ps16InBuf,
        s8UseModeFile,
       pfFileModes, psEncConfig->pps8APPEModeStr, psEncConfig-
        >u8NumFrameToEncode)) > 0)
{
    /* Find out how may frame were actually read */
    psEncConfig->u8NumFrameToEncode = (s16Size/L_FRAME);
    if (psEncConfig->u8NumFrameToEncode != 0)
    {
        /* increment frame number */
        s32Frame = s32Frame + psEncConfig->u8NumFrameToEncode;
        /* initialize output buffer */
        if (psEncConfig->u8BitStreamFormat == NBAMR_ETSI)
        {
```

```
            for (s16Counter = 0; s16Counter < (psEncConfig-
              >u8NumFrameToEncode*SERIAL_FRAMESIZE);
                    s16Counter++)
                ps16OutBuf[s16Counter] = 0;
        }
        else
        {
            for (s16Counter = 0; s16Counter <
         (psEncConfig-
        >u8NumFrameToEncode*((NBAMR_MAX_PACKED_SIZE/2)+
                  (NBAMR_MAX_PACKED_SIZE%2))); s16Counter++)
            {
                ps16OutBuf[s16Counter] = 0;
            }
        }
  /* call encode frame routine */
  eRetVal = eAMREEncodeFrame (psEncConfig, ps16InBuf, ps16OutBuf);
  if (eRetVal != E_NBAMRE_OK)
 {
     /* free all the dynamic memory and exit */
    eRetVal=eAMREEncodeExit(psEncConfig, s8UseModeFile, ps16InBuf,
                ps16OutBuf);
   exit (-1);
}
if (psEncConfig->u8BitStreamFormat == NBAMR_ETSI)
{
   if (psEncConfig->u8BitStreamFormat == NBAMR_ETSI)
   {
       /* write bitstream to output file */
       if ((fwrite (ps16OutBuf, sizeof (NBAMR_S16),
          (psEncConfig->u8NumFrameToEncode*SERIAL_FRAMESIZE),
             pfFileSerial))
          != (psEncConfig->u8NumFrameToEncode*SERIAL_FRAMESIZE))
      {
             exit(-1);
      }
        fflush(pfFileSerial);
  }
  else
 {
     if (psEncConfig->u8BitStreamFormat == NBAMR_IF1IO)
     {
        /* IF1 frame format returns number of bits based on the
        mode used. This includes the header part of 8 * 3 = 24
       bits.
       Following calculation is done to get the number of
       packed bytes to be
       written into the file. 1 is added for the remainder bits.
       */
       *psEncConfig->pu32AMREPackedSize=
                      (*psEncConfig->pu32AMREPackedSize >> 3) + 1;
     }
     for (s16Counter=0; s16Counter < psEncConfig-
                  >u8NumFrameToEncode; s16Counter++)
```

```
          {
                    /* write bitstream to output file */
            if (fwrite
                  ((ps16OutBuf+s16Counter*((NBAMR_MAX_PACKED_SIZE/2)+
                 (NBAMR_MAX_PACKED_SIZE%2))), sizeof (NBAMR_U8),
                  psEncConfig->pu32AMREPackedSize[s16Counter], pfFileSerial)
                      != psEncConfig->pu32AMREPackedSize[s16Counter])
            {
                        exit(-1);
            }
            fflush(pfFileSerial);
    }
          }
        }
    }
 /*********************************************************************
       *Closedown speech coder
*********************************************************************/
    /* free all the dynamic memory and exit */
    eRetVal = eAMREEncodeExit(psEncConfig, s8UseModeFile, ps16InBuf,
                  ps16OutBuf);
    return NBAMR_SUCCESS;
}

/*********************************************************************
* Function: s16AMREReadSpeechSample
*
* Description: This fucntions reads speech sample from buffer or a file
*
* Returns: Number of speech sample read.
*********************************************************************/
NBAMR_S16 s16AMREReadSpeechSample (
            FILE *pfFileSpeech, NBAMR_S16 *ps16InBuf,
            NBAMR_S8 s8UseModeFile, FILE * pfFileModes,
            NBAMR_S8 **pps8ModeStr, NBAMR_U8 u8NumFrame)
{
    NBAMR_S16  s16RetVal = -1;
    NBAMR_S16 s16Index;

   s16RetVal = fread (ps16InBuf, sizeof (NBAMR_S16),
      (u8NumFrame*L_FRAME), pfFileSpeech);
    if (s16RetVal > 0)
    {
        /* Not end of file but read size not same as requested size */
        /* read new mode string from file if required */
        if (s8UseModeFile)
        {
            NBAMR_S32 s32Result;
            for (s16Index =0; s16Index<u8NumFrame; s16Index++)
            {
                pps8ModeStr[s16Index] = (NBAMR_S8
                              *)alloc_fast(10*sizeof(NBAMR_S8*));
             if ((s32Result =
                   s32AMREReadMode(pfFileModes,&pps8ModeStr[s16Index]))
```

```
                        == EOF)
                {
                 return s16RetVal;
                }
                else if (s32Result == 1)
                {
                     return s16RetVal;
                 }
            }
        }
    }
    else if (feof(pfFileSpeech) != NULL)
    {
        /* end of file reached */
        s16RetVal = 0;
    }
    else
    {
        /* error in reading */
        s16RetVal = -1;
     }
    return s16RetVal;
}

/*********************************************************************
*
* Example Implementation of s32AMREReadMode function
*
* Description:
* This function reads mode string from user supplied mode file.
*
* Returns: 0 -> SUCCESS
*          1 -> FAILURE
*          EOF -> End of file (defined as -1 in  stdio.h)
*
* Arguments: pfFileModes -> pointer to mode file
*            pps8ModeStr -> pointer to mode string
*
*********************************************************************/
NBAMR_S32 s32AMREReadMode(FILE *pfFileModes, NBAMR_S8 **pps8ModeStr)
{
    if (fscanf(pfFileModes, "%9s\n", *pps8ModeStr) != 1)
    {
        if (feof(pfFileModes))
        {
            return EOF;
        }
        {
            return NBAMR_FALSE;
        }
    }
    return NBAMR_SUCCESS;
}
```

```
/*************************************************************************
*
* Example implementation of eAMREProcessCmdLineOptions function
*
* Description:  This function processes command line options and
*
* Returns: eAMREReturnType
*************************************************************************/
eAMREReturnType eAMREProcessCmdLineOptions (
           NBAMR_S32 s32Argc, NBAMR_S8 *ps8Argv[],
           NBAMR_S16 *ps16DtxFlag, NBAMR_S8  **pps8ModeStr,
           NBAMR_S8 *ps8UseModeFile, NBAMR_U8 *pu8BitStreamFormat,
           NBAMR_U8 *pu8NumFrame, FILE **ppfFileSpeech,
           FILE **ppfFileSerial, FILE **ppfFileModes
           )
{
    NBAMR_S8 *ps8FileName = NULL;
    NBAMR_S8 *ps8ModeFileName = NULL;
    NBAMR_S8 *ps8SerialFileName = NULL;

    /* Initialize mode file flag */
    *ps8UseModeFile = 0;
    /* Process command line options */
    while (s32Argc > 1)
    {
        if (strcmp((char *)ps8Argv[1], "-dtx1") == 0)
        {
            *ps16DtxFlag = NBAMR_VAD1;
        }
        else if (strcmp((char *)ps8Argv[1], "-dtx2") == 0)
        {
            *ps16DtxFlag = NBAMR_VAD2;
        }
        else if (strcmp((char *)ps8Argv[1], "-mms") == 0)
        {
            *pu8BitStreamFormat = NBAMR_MMSIO;
        }
        else if (strcmp((char *)ps8Argv[1], "-if1") == 0)
        {
            *pu8BitStreamFormat = NBAMR_IF1IO;
        }
        else if (strcmp((char *)ps8Argv[1], "-if2") == 0)
        {
            *pu8BitStreamFormat = NBAMR_IF2IO;
        }
        else if (strncmp((char *)ps8Argv[1], "-numframe=", 10) == 0)
        {
            *pu8NumFrame = (NBAMR_U8)(*(ps8Argv[1]+10));
            *pu8NumFrame = *pu8NumFrame - '0';
            if ((*pu8NumFrame < 1) || (*pu8NumFrame > 255))
            {
                return E_NBAMRE_INVALID_ENCODER_ARGS;
            }
        }
```

```
    else if (strncmp((char *)ps8Argv[1], "-modefile=", 10) == 0)
    {
        *ps8UseModeFile = 1;
        ps8ModeFileName = ps8Argv[1]+10;
    }
    else
    {
        break;
    }
    s32Argc--;
    ps8Argv++;
}
if ((*pu8BitStreamFormat != NBAMR_MMSIO) && (*pu8BitStreamFormat !=
        NBAMR_IF1IO)
   && (*pu8BitStreamFormat != NBAMR_IF2IO))
{
}
/* Check number of arguments */
if ((s32Argc != 4 && !(*ps8UseModeFile)) || (s32Argc != 3 &&
  (*ps8UseModeFile)))
{
    return E_NBAMRE_ERROR;
}
/* Open mode file or convert mode string */
if ((*ps8UseModeFile))
{
    /* allocate the memory for modeString */
    *pps8ModeStr = alloc_fast (10*sizeof(NBAMR_S8));
    ps8FileName = ps8Argv[1];
    ps8SerialFileName = ps8Argv[2];
    /* Open mode control file */
    if (strcmp((char *)ps8ModeFileName, "-") == 0)
    {
        *ppfFileModes = stdin;
    }
    else if ((*ppfFileModes = fopen ((char *)ps8ModeFileName,
        "rt")) == NULL)
    {
        return E_NBAMRE_ERROR;
    }
 }
else
{
    *pps8ModeStr = ps8Argv[1];
    ps8FileName = ps8Argv[2];
    ps8SerialFileName = ps8Argv[3];
}

/* Open speech file and result file (output serial bit stream) */
if (strcmp((char *)ps8FileName, "-") == 0)
{
    *ppfFileSpeech = stdin;
}
```

```
    else if ((*ppfFileSpeech = fopen ((char *)ps8FileName, "rb")) ==
NULL)
    {
          return E_NBAMRE_ERROR;
    }

    if (strcmp((char *)ps8SerialFileName, "-") == 0)
    {
        *ppfFileSerial = stdout;
    }
    else if ((*ppfFileSerial = fopen ((char *)ps8SerialFileName, "wb"))
            == NULL)
    {
        return E_NBAMRE_ERROR;
    }
    return E_NBAMRE_OK;
}
```

3.2 Example calling routine for NB-AMR Decoder

Example calling guidelines for calling the NB-AMR decoder is given below.

Note: This code may not compile if used as is.

```
/*****************************************************************
/* Include Files */

*****************************************************************/
#include <stdio.h>  /* for declaration of FILE */
#include <string.h>
#include <stdlib.h>
#include "nb_amr_dec_api.h"
#include "nbamr_dectest.h"

int main (NBAMR_S32 s32Argc, NBAMR_S8 *ps8Argv[])
{
    NBAMR_S16  *ps16InBuf = NULL;        /*pointer to input speech
                                          samples*/
    NBAMR_S16  *ps16OutBuf = NULL;       /* pointer to output (encoded)
                                          buffer */

    sAMRDMemAllocInfoSubType *psMem;   /*pointer to encoder sub-memory
                                          */
    sAMRDDecoderConfigType   *psDecConfig; /* Pointer to encoder config
                                              structure */
    eAMRDReturnType  eRetVal;     /* local variable */
    NBAMR_S16  s16NumReqs;
    NBAMR_S16  s16Counter;
    NBAMR_U32 s32Frame;

    NBAMR_U8  s8MagicNumber[8];
    NBAMR_U8  u8Mode;
    NBAMR_U16 u16DataSize = 0;
```

```
/* size of packed frame for each mode */
NBAMR_S16 gas16PackedCodedSizeMMS[]=
{
    13, 14, 16, 18, 20, 21, 27, 32,
    6,  0,  0,  0,  0,  0,  0,  1
};
/* size of packed frame for each mode */
NBAMR_S16 gas16PackedCodedSizeIF1[]=
{
    15, 16, 18, 20, 22, 23, 29, 34,
    8,  0,  0,  0,  0,  0,  0,  1
};
/* size of packed frame for each mode */
NBAMR_S16 gas16PackedCodedSizeIF2[]=
{
    13, 14, 16, 18, 19, 21, 26, 31,
    6,  0,  0,  0,  0,  0,  0,  1
};

FILE *pfFileSyn = NULL;
FILE *pfFileSerial = NULL;
NBAMR_U8 u8BitStreamFormat = NBAMR_ETSI; /* default file format */
NBAMR_U8 u8NumFrame = 1;    /* Default number of frame */

NBAMR_S16 *ps16In = NULL;   /* local variable */

/* Get the version info of the sign-on NB AMR */
fprintf(stderr,"Running %s\n",eAMRDVersionInfo());

eRetVal = eAMRDProcessCmdLineOptions (s32Argc, &ps8Argv[0],
                    &u8BitStreamFormat,
                    &u8NumFrame, &pfFileSerial, &pfFileSyn);
if (eRetVal != E_NBAMRD_OK)
{
    exit(1);
}

/* allocate fast memory for encoder config structure */
psDecConfig=(sAMRDDecoderConfigType
  *)alloc_fast(sizeof(sAMRDDecoderConfigType));

if (psDecConfig == NULL)
{
    return NBAMR_FAILURE;
}

psDecConfig->pvAMRDDecodeInfoPtr = NULL;
psDecConfig->pu8APPDInitializedDataStart = NULL;

psDecConfig->u8RXFrameType = 0; /* Encoded with frame type set to
                                   TXType */

psDecConfig->u8BitStreamFormat = u8BitStreamFormat;
```

```
psDecConfig->u8NumFrameToDecode = u8NumFrame;

/* initialize decoder app config base pointers */
  for(s16Counter = 0; s16Counter < NBAMR_MAX_NUM_MEM_REQS;
        s16Counter++)
  {
    (psDecConfig-
  >sAMRDMemInfo.asMemInfoSub[s16Counter].pvAPPDBasePtr)
         = NULL;
}

/* Find decoder memory requiremet */
eRetVal = eAMRDQueryMem(psDecConfig);

if (eRetVal != E_NBAMRD_OK)
{
    /* de-allocate memory allocated for encoder config */
    mem_free(psDecConfig);
    return NBAMR_FAILURE;
}
/* Number of memory chunk requested by the encoder */
  s16NumReqs = psDecConfig->sAMRDMemInfo.s32AMRDNumMemReqs;

/* allocate memory requested by the encoder*/
  for(s16Counter = 0; s16Counter < s16NumReqs; s16Counter++)
  {
     psMem = &(psDecConfig-
        >sAMRDMemInfo.asMemInfoSub[s16Counter]);
        if (psMem->s32AMRDMemTypeFs == NBAMR_FAST_MEMORY)
        {
        psMem->pvAPPDBasePtr = alloc_fast(psMem->s32AMRDSize);
        if (psMem->pvAPPDBasePtr == NULL)
        {
            mem_free(psDecConfig);
            return NBAMR_FAILURE;
        }
        }
        else
        {
        psMem->pvAPPDBasePtr = alloc_slow(psMem->s32AMRDSize);
        if (psMem->pvAPPDBasePtr == NULL)
        {
            mem_free(psDecConfig);
            return NBAMR_FAILURE;
        }
        }
  }
  /* Call decoder init routine */
  eRetVal = eAMRDDecodeInit (psDecConfig);
  if (eRetVal != E_NBAMRD_OK)
  {
    /* free all the dynamic memory and exit */
    eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf, ps16OutBuf);
```



```
    return NBAMR_FAILURE;
  }
  /* allocate memory for output buffer (unpacked) */
  if ((ps16OutBuf = alloc_fast (
  (psDecConfig->u8NumFrameToDecode*L_FRAME) * sizeof (NBAMR_S16)))
       == NULL)
  {
    /* free all the dynamic memory and exit */
    eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf, ps16OutBuf);
    return NBAMR_FAILURE;
  }
  if (u8BitStreamFormat == NBAMR_ETSI)
  {
    if ((ps16InBuf = alloc_fast(
        (psDecConfig->u8NumFrameToDecode*SERIAL_FRAMESIZE)*
                          sizeof (NBAMR_S16))) == NULL)
    {
        /* free all the dynamic memory and exit */
        eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf,
                    ps16OutBuf);
        return NBAMR_FAILURE;
    }
}
else
{
    /* it is MMS or IF1 or IF2 format */
    if ((ps16InBuf = alloc_fast(
              (psDecConfig->u8NumFrameToDecode*
      ((NBAMR_MAX_PACKED_SIZE/2)+(NBAMR_MAX_PACKED_SIZE%2)))*sizeof
              (NBAMR_S16))) == NULL)
    {
        /* free all the dynamic memory and exit */
        eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf,
              ps16OutBuf);
        return NBAMR_FAILURE;
    }
    ps16In = ps16InBuf;
}
  /*************************************************************
   * Decode speech frame till the end
   *************************************************************/
  s32Frame = 0;
if (psDecConfig->u8BitStreamFormat == NBAMR_ETSI)
{
    while ((s16AMRDReadSerialData(ps16InBuf,pfFileSerial,
           (psDecConfig->u8NumFrameToDecode*SERIAL_FRAMESIZE))) ==
            (psDecConfig->u8NumFrameToDecode*SERIAL_FRAMESIZE))
    {
        s32Frame=s32Frame+psDecConfig->u8NumFrameToDecode;
        /* call encode frame routine */
        eRetVal=eAMRDDecodeFrame (psDecConfig, ps16InBuf,
              ps16OutBuf);
        if (eRetVal != E_NBAMRD_OK)
        {
```

```
            /* free all the dynamic memory and exit */
            eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf,
              ps16OutBuf);
            exit (-1);
        }
        /* write synthesized speech to file */
        if (fwrite (ps16OutBuf, sizeof (NBAMR_S16),
              (psDecConfig->u8NumFrameToDecode*L_FRAME), pfFileSyn)
              != (psDecConfig->u8NumFrameToDecode*L_FRAME))
        {
        }
        fflush(pfFileSyn);
    }
}
else    /* MMS or IF1 or IF2 format */
{
    if (psDecConfig->u8BitStreamFormat == NBAMR_MMSIO)
    {
        s16AMRDReadPackedData((NBAMR_U8 *)s8MagicNumber,
        pfFileSerial,strlen(NBAMR_MAGIC_NUMBER));
        if (strncmp((const char *)s8MagicNumber,
                    NBAMR_MAGIC_NUMBER,
                    strlen(NBAMR_MAGIC_NUMBER)))
        {
            eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf,
              ps16OutBuf);
            return E_NBAMRD_INVALID_BITSTREAM;
        }
    }
    while((feof(pfFileSerial)) == NULL)
    {
        for (s16Counter=0; s16Counter<psDecConfig-
                    >u8NumFrameToDecode;s16Counter++)
        {
         ps16InBuf = ps16In +
                    (s16Counter*((NBAMR_MAX_PACKED_SIZE/2)
                  +(NBAMR_MAX_PACKED_SIZE%2)));
           if ((s16AMRDReadPackedData((NBAMR_U8 *)ps16InBuf,
                    pfFileSerial, 1)) == -1)
            {
                eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf,
                    ps16OutBuf);
                return E_NBAMRD_INPUT_READERROR;
            }
            if (psDecConfig->u8BitStreamFormat == NBAMR_MMSIO)
            {
                u8Mode = (NBAMR_U8) (0x0F & (*(NBAMR_U8 *)ps16InBuf
                    >> 3));
                u16DataSize = gas16PackedCodedSizeMMS[u8Mode]-1;
            }
            else if (psDecConfig->u8BitStreamFormat == NBAMR_IF1IO)
            {
                u8Mode = (NBAMR_U8) (0x0F & (*(NBAMR_U8 *)ps16InBuf
                    >> 4));
```

```
                    u16DataSize = gas16PackedCodedSizeIF1[u8Mode]-1;
                }
                else if (psDecConfig->u8BitStreamFormat == NBAMR_IF2IO)
                {
                    u8Mode = (NBAMR_U8) (0x0F & (*(NBAMR_U8
                        *)ps16InBuf));
                    u16DataSize = gas16PackedCodedSizeIF2[u8Mode]-1;
                }
                else
                {
                    u16DataSize = 0;
                    /* invalid bitstream format */
                    eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf,
                        ps16OutBuf);
                    return E_NBAMRD_INVALID_DECODER_ARGS;
                }
                if((s16AMRDReadPackedData ((NBAMR_U8 *)((NBAMR_U8
                  *)(ps16InBuf)+1),pfFileSerial, u16DataSize)) == -1)
                {
                         eRetVal = eAMRDDecodeExit (psDecConfig,
                        ps16InBuf, ps16OutBuf);
                    return E_NBAMRD_INPUT_READERROR;
                }
            }
            ps16InBuf = ps16In;
            s32Frame=s32Frame+psDecConfig->u8NumFrameToDecode;

            /* call decode frame routine */
            eRetVal=eAMRDDecodeFrame (psDecConfig, ps16InBuf,
                  ps16OutBuf);
            if (eRetVal != E_NBAMRD_OK)
            {
                /* free all the dynamic memory and exit */
                eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf,
                  ps16OutBuf);
                exit (-1);
            }
            /* write synthesized speech to file */
            if (fwrite (ps16OutBuf, sizeof (NBAMR_S16), (psDecConfig-
                  >u8NumFrameToDecode*L_FRAME),
               pfFileSyn) != (psDecConfig->u8NumFrameToDecode*L_FRAME))
            {
            }
            fflush(pfFileSyn);
        }   /* end of while loop */
    } /* end of else */

   /*************************************************************
    *Closedown NB-AMR decoder
    *************************************************************/
  /* free all the dynamic memory and exit */
    eRetVal = eAMRDDecodeExit (psDecConfig, ps16InBuf, ps16OutBuf);
    return NBAMR_SUCCESS;
}
```

```
/*****************************************************************
* Function: s16AMRDReadSerialData
*
* Description: This function reads decoded speech data of size  250
*           words
*
* Returns: Size of read data
*
* Arguments: ps16InBuf -> pointer to input buffer
*            pfFileSerial -> pointer to serial file
*****************************************************************/
NBAMR_S16 s16AMRDReadSerialData (NBAMR_S16 *ps16InBuf, FILE
*pfFileSerial,NBAMR_U16 u16DataSize)
{
    NBAMR_S16 s16NumData = -1;
    if (fread (ps16InBuf, sizeof (NBAMR_S16), u16DataSize,
      pfFileSerial) ==  u16DataSize)
      {
        s16NumData = u16DataSize;
    }
    return s16NumData;
}


/*****************************************************************
* Function: s16AMRDReadPackedData
*
* Description: This function reads encoded speech data of requested size
in case of MMS, IF1 or IF2 file format
*
* Returns: Size of read data
*****************************************************************/
NBAMR_S16 s16AMRDReadPackedData (
                  NBAMR_U8 *pu8InBuf,
                  FILE  *pfFileSerial,
                  NBAMR_U16 u16DataSize)
{
    NBAMR_S16 s16NumData = -1;

    if ((fread (pu8InBuf, sizeof (NBAMR_U8), u16DataSize,
            pfFileSerial)) == u16DataSize)
    {
        s16NumData = u16DataSize;
    }
    return s16NumData;
}

/*****************************************************************
* Function: eAMRDProcessCmdLineOptions
*
* Description: Function process decoder command line options
*
* Returns: eAMRDReturnType
```

```
*
* Arguments: s32Argc -> Number of command line option
*            ps8Argv -> pointer to an array of command line arguments
*            ppfFileSerial -> pointer to pointer to file
*                                 serial file name
*            ppfFileSyn -> pointer to pointer to synthesis file name
*****************************************************************/
eAMRDReturnType eAMRDProcessCmdLineOptions (
            NBAMR_S32 s32Argc, NBAMR_S8 *ps8Argv[],
            NBAMR_U8 *pu8BitStreamFormat, NBAMR_U8 *pu8NumFrame,
            FILE **ppfFileSerial, FILE **ppfFileSyn
            )
{
    NBAMR_S8  *ps8FileName = NULL;
    NBAMR_S8  *ps8SerialFileName = NULL;

    while (s32Argc > 1)
    {
        if (strcmp((char *)ps8Argv[1], "-mms") == 0)
        {
            *pu8BitStreamFormat = NBAMR_MMSIO;
        }
        else if (strcmp((char *)ps8Argv[1], "-if1") == 0)
        {
            *pu8BitStreamFormat = NBAMR_IF1IO;
        }
        else if (strcmp((char *)ps8Argv[1], "-if2") == 0)
        {
            *pu8BitStreamFormat = NBAMR_IF2IO;
        }
        else if (strncmp((char *)ps8Argv[1], "-numframe=", 10) == 0)
        {
            *pu8NumFrame = (NBAMR_U8)(*(ps8Argv[1]+10));
            *pu8NumFrame = *pu8NumFrame - '0';
            if ((*pu8NumFrame < 1) || (*pu8NumFrame > 255))
            {
                return E_NBAMRD_INVALID_DECODER_ARGS;
            }

        }
        else
        {
            break;
        }
        s32Argc--;
        ps8Argv++;
    }
    /* check command line options */
    if (s32Argc != 3)
    {
        return E_NBAMRD_INVALID_DECODER_ARGS;
    }
    ps8SerialFileName = ps8Argv[1];
    ps8FileName = ps8Argv[2];
```

```
    if ((*ppfFileSerial = fopen ((char *)ps8SerialFileName, "rb")) ==
            NULL)
    {
        return E_NBAMRD_ERROR;
    }
    if ((*ppfFileSyn = fopen ((char *)ps8FileName, "wb")) == NULL)
    {
        return E_NBAMRD_ERROR;
    }
    return E_NBAMRD_OK;
}
```