08-6354-SIS-ZCH66

4/1/2008

5.4

Application Programmers Interface for H.264/AVC Decoder

ABSTRACT:

Application Programmers Interface for H.264/AVC Decoder

KEYWORDS:

Multimedia codecs, H.264, AVC, MPEG

Approved:

Wang Zening

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	23-Jul-2004	Murali	Initial Draft
0.2	11-Aug-2004	Murali	Review Rework
0.3	24-Sep-2004	Chandra/Murali	Review Work
0.4	18-Oct-2004	Chandra/Murali	Review Rework
0.5	18-Nov-2004	Chandra/Murali	Review Rework
1.0	07-Dec-2004	Chandra	Formatted for release 1.0
1.1	21-Dec-2004	Chandra	Revised document for call back changes and formatted text
2.0	14-Apr-2005	Chandra	Changed version number and updated history
2.1	21-Nov-2005	Raja	Updated the APIs and Callback functions to handle the application specific pointer.
3.0	06-Feb-2006	Lauren Post	Using new format
3.1	1-Apr-2006	Manoj Arvind	Updated API structures
3.2	09-May-2006	Purusothaman	Added the details of the Logger Function and its API.
3.3	09-June-2006	Dheeraj C Kuchangi	Updated API after clean up.
3.4	24-July-2006	Dheeraj C Kuchangi	Updated the APIs for additional output format(UYVY) feature
3.5	4-Sep-2006	Madhu Kumar	Updated APIs and API structures after implementing Annexure C
4.0	21-Sep-2006	Vineet Golchha	Updated APIs to avoid memory copy within library
4.1	26-Sep-2006	Vineet Golchha	Updated changes pertaining to the new VOB structure
4.2	27-Sep-2006	Manoj Arvind	Updated API structure for PAF and cleaned up the sample application code.
4.3	28-Sep-2006	Manoj Arvind	Updated comments
5.0	23-Apr-2007	Wang Zening	1. add pre-fetch callback (2.1.7, 2.1.8) 2. Update return type (2.2.1), E_NO_PICTURE_PAR_SET_NAL and E_NO_SEQUENCE_PAR_SET_NAL 3. add pre-fetch callback register function (2.3.3)

			4. update section 3.1,section 3.3 5. Add more comments and a appendix (6.1)
5.1	31-Oct-2007	Li Xianzhong	Add error handling, update API return code E_AVCD_BAD_DATA
5.2	8-11-2007	Wang Zening	Add DR change
5.3	14-Nov-2007	Li Xianzhong	Change API return code type
5.4	1-Apr-2008	Li Xianzhong	Add API for buffer release
5.5	16-Apr-2008	Li Xianzhong	Add API for query physical memory
5.6	13-Jun-2008	Li Xianzhong	Add API for query codec version and demo protection information

Table of Contents

1 Introduction

1.1 Purpose

This document gives the application programmer's interface to H.264 baseline / MPEG-4 Part 10 decoder library.

1.2 Scope

This document does not give the detailed implementation of the decoder. It only explains the APIs and data structures exposed to the application developer for using the decoder library

1.3 Audience Description

The reader is expected to have basic understanding of video processing and video coding standards.

1.4 References

1.4.1 Standards

1. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC), JVT-G050r1.doc
2. Proposed Draft AVC|H.264 Conformance Spec, JVT-J011.doc

1.4.2 Freescale Multimedia References

3. H.264 Decoder Application Programming Interface – h264_dec_api.doc
4. H.264 Decoder Requirements Book - h264_dec_reqb.doc
5. H.264 Decoder Test Plan - h264_dec_test_plan.doc
6. H.264 Decoder Release notes - h264_dec_release_notes.doc
7. H.264 Decoder Test Results – h264_dec_test_results.doc
8. H.264 Decoder Performance Results – h264_dec_perf_results.doc
9. H.264 Decoder Interface Header – avcd_dec_api.h
10. H.264 Decoder Application Code – decoder.c

1.5 Definitions, Acronyms, and Abbreviations

TERM/ACRONYM	DEFINITION
AVC	Advanced Video Coding
API	Application Programming Interface
ARM	Advanced RISC Machine
DPB	Decoded Picture Buffer
FSL	Freescale
ISO	International Standards Organization
ITU	International Telecommunication Union
MPEG	Moving Pictures Expert Group
NAL	Network Abstraction Layer
PAF	Parallelization Across Frames
RVDS	ARM RealView Development Suite
UNIX	Linux PC x/86 C-reference binaries

1.6 Document Location

docs/h264_dec

2 API Description

This section describes the data structures followed by an example usage of the H.264 video decoder. The decoder supports all the levels in the baseline profile, the user can however set a default level (information used during memory allocation and in the decoder library) by setting the parameter AVCD_DEFAULT_LEVEL_SUPPORT to the required level. The various levels and their constraints are specified in [1.4.2-2].

2.1 Data Structures

This section describes the data structures used in the decoder interface.

2.1.1 sAVCDecoderConfig

This is the main data structure which should be passed to all the decoder functions. The definition of the structure is given below.

```
typedef struct
{
   long                        s32NumBytes;
   unsigned short              s32NalType;
   void                        *pvInBuffer;
   int                         s32FrameNumber;
   long                        s32InBufferLength;
   void                        *pvAVCData;
   sAVCDConfigInfo             sConfig;
   sAVCDMemAllocInfo           sMemInfo;
   sAVCDYCbCrStruct            sFrameData;
   int                         paf;
   unsigned char               u8Status;
   void                        *pAppContext;
   int   (*cbkAVCDBufRead)   ( unsigned char *pu8Buf,
                               int s32BufLen,
                               int *s32Last,
                               void *pAppContext);
} sAVCDecoderConfig;
```

Description of structure *sAVCDecoderConfig*

s32NumBytes

Number of bytes in input buffer. This parameter is to be filled in by the application that is passing the bitstream to the decoder.

S32NalType

Type of the NAL filled by the application

pvInBuffer

Pointer to Input Bit stream

s32FrameNumber

This denotes the frame that has been decoded recently.

s32InBufferLength

Size (in bytes) of input bitstream buffer determined by the application.

pvAVCData

This is an internal video frame context for the decoder and application should not change this.

sConfig

A scalable data structure that provides useful information on the bitstream to the application.

sMemInfo

This is memory information structure. This is described in sAVCDMemAllocInfo [2.1.2].

sFrameData

Reconstructed frame Data (Y, Cb and Cr components).

paf

This flag is set by the library if PAF scheme is enabled.

u8Status

Flag indicating decoding status (complete or incomplete frame decode before copying the reconstructed data to application space). This flag is set to '1' when decoding of a frame is completed. This along with the API return type, notify the application on when to render the output frame.

pAppContext

Application specific data. The codec library will not edit this data. This will be given back to the application during call back.

cbkAVCDBufRead

This is a synchronous call used by the decoder to read the bit stream from the application. More detailed explanation is given later in the Input Buffer Interface [2.1.6]

2.1.2 sAVCDMemAllocInfo

This structures holds the information required for the memory management of the decoder. The decoder memory requirements are passed to the application when *eAVCDQueryMem* and eAVCDReQueryMem is called. The decoder specifies number of memory blocks needed by filling s32NumReqs in this structure. Each valid entry in the array "asMemBlks" describes size and properties of memory blocks required by the decoder. Application shall allocate the memory required by looking at this structure before initializing the decoder.

```
typedef struct
 {
       int              s32NumReqs;
       sAVCDMemBlock    asMemBlks[MAX_NUM_MEM_REQS];
       int              s32MinFrameBufferNum;
 } sAVCDMemAllocInfo;
```

Description of structure *sAVCDMemAllocInfo*

s32NumReqs

Number of memory blocks required. Decoder will set this to required value when e*AVCDQuerymem* function is called.

asMemBlks

Array of memory block structure, for each request defined in *s32NumReqs* application should allocate the memory. MAX_NUM_MEM_REQS is the maximum number of memory chunk requests the decoder can make. Currently it is set to 40.

s32MinFrameBufferNum

A integer number that indicates the minimum frame buffers that need by the decoder during decoding, this value will be used for correctly creating the frame buffer manger. This value will be got after invoking eAVCDReQueryMem ()

2.1.3 sAVCDMemBlock

This describes the memory block details such as size, type, etc.

```
typedef struct
{
    int         s32Size;
    int         s32Align;
    int         s32Type;
    int          s32Priority;
    int          s32SizeDependant;
    int          s32Allocate,
    int          s32Copy,
    int          s32MaxSize;
    void        *pvBuffer;
} sAVCDMemBlock;
```

Description of the structure *sAVCDMemBlock*

s32Size

The size of the memory required.

s32Align

The alignment of the memory block. It can be one of NO_ALIGNMENT, HALF_WORD, or WORD_ALIGNED

s32Type

The Type of memory needed. It can be one of SLOW_SCRATCH, SLOW_STATIC, FAST_SCRATCH, FAST_STATIC

s32Priority

Provides a guideline on the criticality of the memory block for performance. Priority “0” is the highest.

s32SizeDependant

Indication if the parameter depends on size of frame. If this parameter is set then the size of the element specified can dynamically between two consecutive frame decodes. If this is set and the application does not choose to use the maximum size for the element (s32MaxSize) then the application ‘re-queries’ the memory again when there is change in any of the configuration parameters (frames size and/or number of reference frames)

s32Allocate

Indicates that memory is required to be allocated if the application decides not to allocate memory for the worst case. If not set to ‘1’, no memory is allocated.

s32Copy

Indicates that the old contents in the memory needs to be copied before re-allocating the memory.

s32MaxSize

Specifies the maximum possible size for the specified memory element. If application chooses to use this instead of using s32Size then the application does not need to re-query the memory again (occurs due to change of frames sizes and number of reference frames

used by the current decoded frame). This field is not to be used for memory allocation in the current version of the software.

pvBuffer

This will be updated by the application based on the address of the allocated memory.

NOTE Size and alignment are mandatory specifications to the application, where as type and priority helps the application to correctly allocate the memory. The s32MaxSize should not be used for memory allocation in the current version of the software.

2.1.4 sAVCDYCbCrStruct

This Data structure encapsulates the decoded YCbCr buffer.

```
typedef struct
{
    unsigned char       *pu8y,*pu8cb,*pu8cr;
    long                s32FrameNumber;
    short               s16FrameWidth;
    short               s16FrameHeight;
    short               s16Xsize;
    short               s16CxSize;
    eAVCDOutputFormat   eOutputFormat;
    int                 cropLeft_display;
    int                 cropTop_display;

} sAVCDYCbCrStruct;
```

Description of the structure *sAVCDYCbCrStruct*

pu8Y

Pointer to Output Y buffer (see figure 1)

pu8Cb

Pointer to Output Cb buffer

pu8Cr

Pointer to Output Cr buffer

s32FrameNumber

This element is used as a flag when DPB_FIX_APP is not defined with PAF scheme enabled

s16FrameWidth

Frame width of the output frame.

s16FrameHeight

Frame height of the ouput frame.

s16Xsize

X dimension of y buffer. Should be greater than or equal to frameWidth. In case of UYVY output this should be equal to the row buffer size (2*row pixels). This can be used to remove the padded information and for cropping the video frame.

s16CxSize

X dimension of cb/cr buffer. Should be greater than or equal to frameWidth/2. Not used for UYVY output format.

eOutputFormat

Output format required from the library. This is a mandatory field that needs to be set during the library initialization. [sec 2.2.3]

cropLeft_display

CropLeft_display provides information on the number of bytes to be cropped on the left of the decoded video frame. This value is used by the test application to calculate the pu8y, pu8cb and pu8cr pointers under the E_AVCD_420_PLANAR_PADDED output format.

cropTop_display

CropTop_display provides information on the number of bytes to be cropped on the top of the decoded video frame. This value is used by the test application to calculate the pu8y, pu8cb and pu8cr pointers under the E_AVCD_420_PLANAR_PADDED output format.

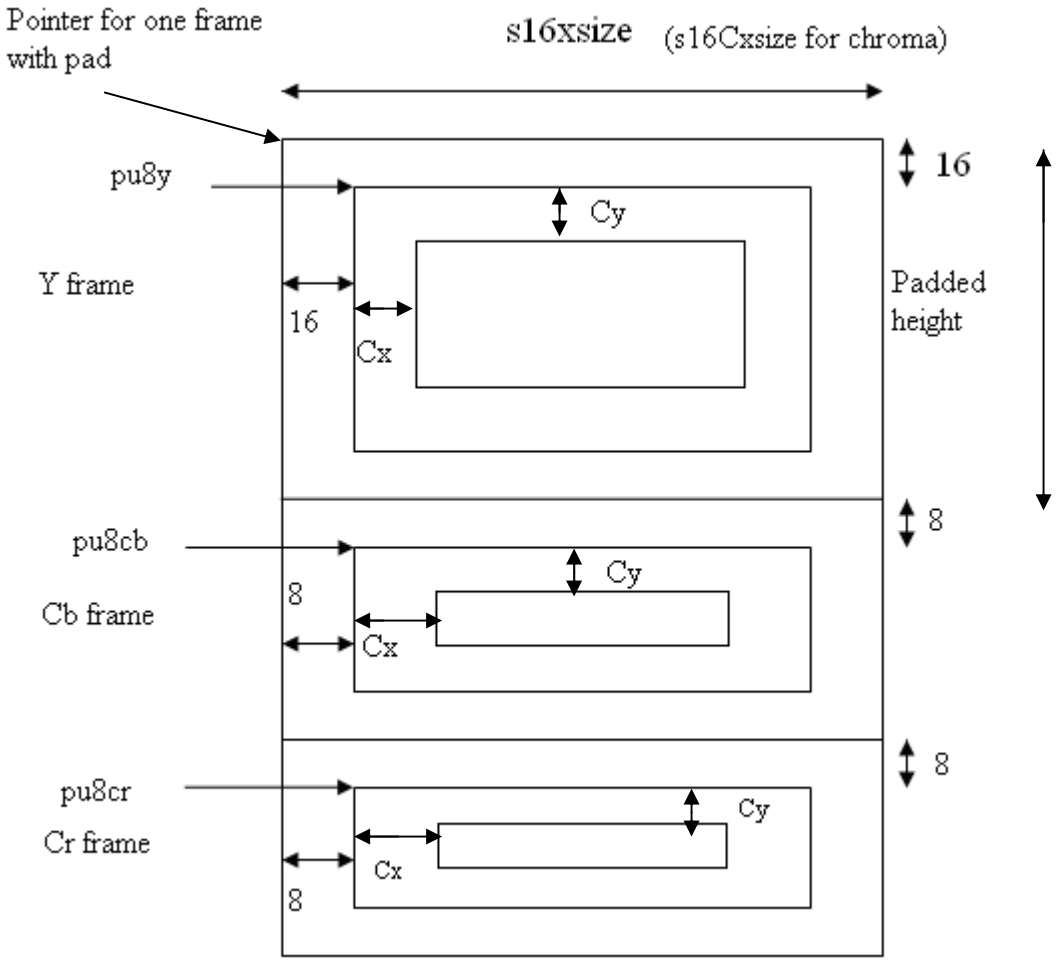

Figure 1: Storage format and pointers one output frame

Cx = cropLeft_display * 2 (for luma) and Cx = cropLeft_display (for Cb and Cr)
Cy = cropTop_display * 2 (for luma) and Cy = cropTop_display (for Cb and Cr)

In the above diagram, the value 16 vertically and horizontally specifies the pad. Similarly the value 8 for the Cb and Cr components.

To calculate the pointer with the pad(as shown in figure) the following relations can be used:

```
f->pu8y - = ((16*f->s16Xsize) + 16)
f->pu8cb - = ((8* (f->s16Xsize>>1) ) + 8)
```

For calculating the padded height parameter (as indicated in the figure), after step computing the above, padded_height = (f->pu8cb - f->pu8y) / f->s16Xsize.

2.1.5 sAVCDConfigInfo

This Data structure contains width and height, this could be expanded in future to contain other parameters when needed.

```
typedef struct
{
    short               s16FrameWidth;
    short               s16FrameHeight;
    short               s16NumRefFrames;
    short               s16Level;
   unsigned int         u32MaxDPB;
}sAVCDConfigInfo;
```

Description of the structure *sAVCDConfigInfo*

s16FrameWidth

Frame width of the current decoded frame

s16FrameHeight

Frame height of the current decoded frame

s16NumRefFrames

Number reference frame used by the current decoded frame

s16Level

Level defined by the current bitstream

u32MaxDPB

Max DPB supported for the corresponding level

2.1.6 Input buffer interface

cbkAVCDBufRead is a synchronous call used by the decoder to read the bit stream from the application. This function is called by the decoder in e*AVCDecodeNALUnit* functions, when it runs out of current bit stream buffer. This function is not part of the library and has to be supplied by the user of the decoder library. Application developer has complete control in implementing this API based on the system requirement.

Prototype:

```
int (*cbkAVCDBufRead) ( unsigned char *pu8Buf,
                        int s32BufLen,
                        int* s32Last,
                        void* pAppContext);
```

Arguments:

- *pu8Buf* Pointer to the buffer.
- *s32BufLen* Number of bytes requested by the decoder.
- *s32Last* Set to ‘1’ by the application if the number of bytes for a NAL unit that is being decoded is exhausted. If more data is present after the call back it is set to ‘0’.
- *pAppContext* The application specific data which is given by the application during the decode call and is give back to the application during this callback.

Return value:

Returns the size of the buffer copied else return -1 for failure

2.1.7 Pre-fetch NAL interface

For accelerating the decoder, a Pre-fetch NAL data scheme is used. It's an additional scheme which means decoder's performance will be improved if Application utilized this scheme. And if Application didn't adopt this scheme the decoder functionality will be kept as same except performance will be degraded a little.

There are 2 types of call back function pointer are used:

- cbkAVCDPrefetchNAL
- cbkAVCDLengthSetter

cbkAVCDPrefetchNAL is a function pointer type whose instance is a synchronous call used by the decoder to pre-fetch the encoded bit stream from the application. This function is called by the decoder when it runs out of current bit stream buffer.
By using this function, decoder can reuse the NAL buffer which is provided by application directly. Comparing with the *cbkAVCDBufRead* method which is described in section 2.1.6, this pre-fetch method can avoid memory copy inside decoder.

This function is not part of the library and has to be supplied by the user of the decoder library. Application developer has complete control in implementing this API based on the system requirement.

Prototype:
```
typedef int (*cbkAVCDPrefetchNAL)(unsigned char **ppbuf, int *len);
```

Arguments:

- ***ppbuf*** Pointer of Pointer to the NAL buffer which is provided by the application.
- ***len*** Number of bytes requested by the decoder.

Return value:
Returns 0 for success or return -1 for failure

cbkAVCDLengthSetter is a function pointer type whose instance is a synchronous call used by the decoder to inform Application the length change of NAL buffer after decoder pre-fetched NAL. This function is not part of the library and has to be supplied by the user of the decoder library. Application developer has complete control in implementing this API based on the system requirement.

Prototype:
```
typedef void (*cbkAVCDLengthSetter)(int len_after_destuff);
```

Arguments:

- ***len_after_destuff*** the buffer length after destuff process which is performed after NAL data fetching

Return value:

N/A

Instances of these 2 function pointer type will be registered into decoder via an API functions named *eAVCDSetNALFuncs*. This step should be performed after application initialized decoder via *eAVCDInitQueryMem*. The detailed procedure will be provided in later section.
Example implementations of these 2 pre-fetch functions are provided in Appendix (section 0).

2.1.8 sAVCDNAL_FUNCs

This Data structure contains 2 functions that described in the previous section 2.1.7, and will be used to register application implemented functions into decoder.

```
typedef struct _NAL_FUNC
{
      cbkAVCDPrefetchNAL NALFetcher;
      cbkAVCDLengthSetter NALLengthSetter;
}sAVCDNAL_FUNCs;
```

Description of the structure *sAVCDNAL_FUNCs*

NALFetcher

A function pointer which will be used by decoder to fetch NAL data

NALLengthSetter

A function pointer which will be used by decoder to inform application the buffer length after destuff process which is performed after NAL data fetching

2.2 Enumerations and Typedefs

2.2.1 Library API Return codes

```
typedef enum
{
    //!< Successful Completion
    E_AVCD_NOERROR =  0,
    E_AVCD_INIT,
    E_AVCD_QUERY,
    E_AVCD_SEQ_CHANGE,
    E_AVCD_CHANGE_SERVICED,
    E_AVCD_FF,
    E_AVCD_FLUSH_STATE,

    //!< Recoverable Errors, warnings and information
    E_AVCD_NOT_SUPPORTED,
    E_AVCD_BAD_PARAMETER,
    E_AVCD_BAD_DATA_PTR ,
    E_AVCD_NOMEM,
    E_AVCD_NO_FRAME_BUFFER_CHANGE,
    E_AVCD_FRAME_BUFFER_CHANGE,
    E_AVCD_NO_OUTPUT,
    E_AVCD_NULL_POINTER,
    E_AVCD_BAD_DATA,
    E_AVCD_OUTPUT_FORMAT_NOT_SUPPORTED,
    E_NO_PICTURE_PAR_SET_NAL,
    E_NO_SEQUENCE_PAR_SET_NAL
    E_AVCD_DEMO_PROTECT,
```

```
    //!< Irrecoverable Errors
    E_AVCD_CODEC_TYPE_NOT_SUPPORTED,
    E_AVCD_INVALID_PARAMETER_SET,
    E_AVCD_UNKNOWN_ERROR = 127,
}eAVCDRetType;
```

Return type	Error Type	Comments
E_AVCD_NOERROR	None	The function execution is successful.
E_AVCD_INIT	None	The decoder is initialized and is ready for doing operations.
E_AVCD_QUERY	None	This state refers to that querying for the ***initial*** memory requirement is completed and the application has to allocate the required memory
E_AVCD_SEQ_CHANGE	None	This state is returned when there is a change in any of the configuration parameter that would warrant changes in the memory requirements during the decoding. Application needs to '*re-query*' the memory for re-allocation if the maximum memory for the default level is not allotted.
E_AVCD_CHANGE_ SERVICED	None	This state is returned when '*re-query*' is done and the new memory requirement is returned to the application. The application has to allocate the required memory if hasn't chosen to allocate the maximum memory requirement for the supported level.
E_AVCD_FF	None	This state is referred if the user opts for FF of frame(s). Once an IDR frame is encountered the state automatically switches to E_AVCD_PLAY from this state.
E_AVCD_FLUSH_STATE	None	This state is returned when DPB is full and one or more decoded frames have to be flushed from DPB. The decoder has to be fed with the same NALU as far as this state is returned.
E_AVCD_NOT_ SUPPORTED	Recoverable	Level specified by the bitstream is not supported
E_AVCD_CODEC_TYPE_NO T_SUPPORTED	Irrecoverable	Unsupported codec type
E_AVCD_BAD_ PARAMETER	Recoverable	Invalid parameter(s)
E_AVCD_BAD_DATA_ PTR	Recoverable	Invalid memory for data pointer.
E_AVCD_NOMEM	Recoverable	Not enough memory for decoding
E_AVCD_NO_FRAME_ BUFFER_CHANGE	Recoverable	Information that no buffer changes are required
E_AVCD_FRAME_ BUFFER_CHANGE	Recoverable	Warning that buffer changes occurred and handling required
E_AVCD_NO_OUTPUT	Recoverable	Successful decoding but output not generated
E_AVCD_NULL_POINTER	Recoverable	Buffers are not allocated as requested
E_AVCD_OUTPUT_FORMA T_NOT_SUPPORTED	Recoverable	Output format expected is not supported

E_NO_PICTURE_PAR_SET_NAL	Recoverable	When decoder tries to decode a NAL, decoder found that there is not a Picture Parameter NAL decoded previously. Application can just treat this error as met a E_AVCD_NO_OUTPUT
E_NO_SEQUENCE_PAR_SET_NAL	Recoverable	When decoder tries to decode a NAL, decoder found that there is not a Sequence Parameter NAL decoded previously. Application can just treat this error as met a E_AVCD_NO_OUTPUT
E_AVCD_BAD_DATA	Recoverable	When decoder detects the error which can be handled, it try to decode current NAL, the output frame data is not correct, when the frame is outputted, API return E_AVCD_BAD_DATA, application decide to display the frame or not
E_AVCD_INVALID_PARAMETER_SET	Irrecoverable	Invalid Pic/Seq param set
E_AVCD_UNKNOWN_ERROR	Irrecoverable	Some Unknown error
E_AVCD_DEMO_PROTECT	Recoverable	Only for demo protection version, when the decoded frame count is greater than 9000,codec always return E_AVCD_DEMO_PROTECT

2.2.2 Alignment definitions

```
typedef enum
{
    E_AVCD_BYTE_ALIGN = 0,
    E_AVCD_HALFWORD_ALIGN,
    E_AVCD_THIRDBYTE_ALIGN,
    E_AVCD_WORD_ALIGN,
}eAVCDAlign;
```

Return type	Comments
E_AVCD_BYTE_ALIGN	No alignment required
E_AVCD_HALFWORD_ALIGN	Alignment to the second byte in the word
E_AVCD_THIRDBYTE_ALIGN	Alignment to the third byte in the word
E_AVCD_WORD_ALIGN	Alignment to the word boundary

2.2.3 Output format definitions

This enum specifies the output format required.

```
typedef enum
{
   E_AVCD_420_PLANAR = 0,
   E_AVCD_420_PLANAR_PADDED,
   E_AVCD_422_UYVY
}eAVCDOutputFormat;
```

2.2.4Buffer getter

Since the Direct Rendering is adopted, the decoder will ask to get a new buffer for decoding. A function which is implemented by the application (frame work) will be used to perform getting a frame buffer for decoding.

Prototype:

```
typedef void* (*bufferGetter)( void* /*pvAppContext*/);
```

Arguments:

- Application context

Return value:

- A frame buffer

2.2.5Buffer Rejecter

Since the Direct Rendering is adopted, the decoder will ask to get a new buffer for decoding.
It's possible that the gotten frame buffer may be refused by the decoder. Decoder need to inform the application (framework) that this frame is rejected.
A function which is implemented by the application (frame work) will be used to perform reject ion of a frame buffer.

Prototype:

```
typedef void (*bufferRejecter)( void* /*mem_ptr*/, void*
/*pvAppContext*/);
```

Arguments:

- A rejected frame buffer
- Application context

Return value:

- None

2.2.6Buffer Manager

For clarifying the concept and simplifying the API, we group the 2 function pointer we described above into a structure named DR_BufferManager:

```
typedef struct _AVCD_FrameManager
{
    bufferGetter BfGetter;
    bufferRejecter BfRejector;
}AVCD_FrameManager;
```

2.2.7Deblock Option

Since the IC may or may not has HW deblock function, this H264 decoder offer functions to check and set the deblock options of the decoder.

```
typedef enum
{
      E_AVCD_SW_DEBLOCK = 0,
```

```
    E_AVCD_HW_DEBLOCK,
}eAVCDDeblockOption;
```

Deblock type	Comments
E_AVCD_SW_DEBLOCK	Use Software Deblock
E_AVCD_HW_DEBLOCK	Use Hardware Deblock

2.2.8Buffer Releaser

Since the Direct Rendering is adopted, the decoder is responsible to release an unused buffer got by buffer getter, the application(frame work) will reclaim this buffer from decoder. A function which is implemented by the application (frame work) will be used to perform releasing a frame buffer from decoding.

Prototype:

```
typedef void (*bufferReleaser)( void* /*mem_ptr*/, void* /*pvAppContext*/);
```

Arguments:

- A released frame buffer
- Application context

Return value:

- None

Application Programmer Interface Functions

2.2.9 Query Memory

This is the first function to be called by the application. This function sets the initial memory requirement of the decoder. The decoder does not parse the bit stream to determine the values to populate the memory structures. This call would set the maximum memory for a given level for all the decoder components defined in the memory structure. It also sets a flag (for each of the component) whether the memory requirement is subject to change during the process of decode. It also sets the actual sizes for each of the component. For components where sizes do not change during the actual decode, the actual size and the maximum size are the same, if different the actual sizes are known during '***re-querying***' of the memory explained in 2.3.3. The application allocates the requested blocks of memory using the information from the decoder memory structure.

Prototype:

```
eAVCDecRetType eAVCDInitQuerymem (sAVCDMemAllocInfo *psMemPtr);
```

Arguments:

- psMemPtr Memory requirement structure for the decoder.

Return value:

eAVCDecRetType

Specifies whether assignment of parameters needed for memory allocation was successful or not. Enumeration is described in the above section. Return values are -

E_AVCD_QUERY - Function successful.
Other values - Error

2.2.10 Query Physical Memory Address

This interface is for eLinux BSP only, Since IPU is used to de-block picture, PF driver is called and need implementation in codec library. application will prepare memory,

2.2.11 Initialization

All initializations required for the decoder are done in e*AVCDInitVideoDecoder*. The output format required should be set appropriately. This function must be called before the main decoder functions are invoked.

Prototype:

```
eAVCDecRetType eAVCDInitVideoDecoder (sAVCDecoderConfig *psAVCDec);
```

Arguments:

- psAVCDec Decoder object pointer.

Return value:

eAVCDecRetType

Specifies whether decoder has been successfully initialized or not. Enumeration is described in the above section. Return values are -

E_AVCD_INIT - Function successful.

Other values - Error

2.2.12 ReQuery Memory

This function is called by the application when the decoder detects a change in the configuration (frame size or number of frames) during frame decode. In these scenarios the decoder exits out with the required information (as contained in the configuration information structure in section 2.1.5) and the application reallocates the required memory after calling this routine. If the maximum memory is already allocated after the initial query (for the decoded level), then the application does not need do any reallocation.

Prototype:

```
eAVCDecRetType eAVCDReQueryMem (sAVCDecoderConfig *psAVCDec)
```

Arguments:

- psAVCDec Decoder object pointer.

Return value:

eAVCDecRetType

Specifies whether frames were decoded successfully or not.
Enumeration is described in the above section. Return values are -

E_AVCD_NOERROR - Function successful.
Other values - Error

2.2.13 Register NAL pre-fetch callback functions

This function is optional which means application make choice that if it will use this function. Decoder's performance will be improved if Application utilized this scheme. And if Application didn't adopt this scheme the decoder functionality will be kept as same except performance will be dropt a little.

This function is called by the application after decoder initialization. The detailed procedure is:

- Application should prepare those 2 NAL pre-fetch functions that described in section 2.1.7.
- Then pack these 2 NAL pre-fetch functions into a `sAVCDNAL_FUNCs` structure as described in 2.1.8.
- Invoke this function whit the `sAVCDNAL_FUNCs` structure.

Prototype:

```
void eAVCDSetNALFuncs( sAVCDNAL_FUNCs *pNalFuncs );
```

Arguments:

- pNalFuncs pointer to a structure which is used to pack pre-fetch functions.

Return value:

None

2.2.14 Decode

The main decoder function is *eAVCDecodeNALUnit*. This function decodes the H264 bit stream in the input buffers to generate one frame of decoder output every call. The decoded output

parameters are populated on the structure elements of sAVCDYCbCrStruct. Decoded output buffer pointers are populated on structure elements pu8y, pu8cb and pu8cr. The decoded buffer would have padded video frame. The horizontal size of padded frame is s16Xsize. The actual horizontal size of the video frame is populated on the structure element s16FrameWidth and the actual height is populated on s16FrameHeight. The cropping left and top offset for chroma is populated on cropLeft_display and cropTop_display respectively.

Prototype:

```
eAVCDecRetType eAVCDecodeNALUnit (sAVCDecoderConfig *psAVCDec, unsigned char u8FastForwardFlag);
```

Arguments:

- psAVCDec Decoder object pointer.
- u8FastForwardFlag Fast forward is ON/OFF.

Return value:

eAVCDecRetType

Specifies whether frames were decoded successfully or not.

Enumeration is described in the above section. Return values are -

E_AVCD_NOERROR - Frame Decode successfully completed.

E_AVCD_FLUSH_STATE – Input NALU provided not decoded. One frame data is outputted.

Other values - Error/Warnings/Information (for more details please refer to section 2.2.1.

Note:

When fast forward is set, this API skips to an Intra Refresh Frame in the decoded stream. At the start of video coding layer, the NAL unit type is decoded and examined for the identity of the slice to be encoded. If the slice is not an IDR frame, the NAL unit is not decoded and the control shifts back to the application for getting a new NAL unit. Once an IDR frame is detected it is decoded and the return type is changed from E_AVCD_FF to E_AVCD_PLAY. In this scenario the application should examine the return type of the decoder (assuming application stores previous state) and determines whether to fast forward further or reset the ‘u8FastForwardFlag’ to 0 (for regular decode).

2.2.15 Output Frame

eAVCDGetFrame is an optional API which the application can use to copy the decoded video frame data in to a buffer allocated by the application. The pointers to the buffer allocated by the application must be populated on the structure elements of sAVCDYCbCrStruct (pu8y, pu8cb and pu8cr). This API will copy the cropped video frame to this output buffer. This API need not be used if the application can handle cropping of video data using hardware.

This API will not be invoked when *eAVCDOutputFormat [sec 2.2.3]* is set to *“E_AVCD_420_PLANAR_PADDED”* (see sample test application).

The current release doesnot support the output format *eAVCDOutputFormat E_AVCD_422_UYVY*.

Prototype:

```
void eAVCDGetFrame( sAVCDecoderConfig *psVDec )
```

- psVDec Pointer to the decoder structure which in turn uses the pointers by test application.

2.2.16 Output Flush

This function is to be called after the input stream is completely decoded and before freeing the decoder. This function should be called repeatedly until E_AVCD_NO_OUTPUT is returned. The output frames would be in the psAVCDec->sFrameData and should be displayed before the next *eAVCDecoderFlushAll* call.

Prototype:

```
eAVCDecRetType eAVCDecoderFlushAll (sAVCDecoderConfig *psAVCDec);
```

Arguments:

- psAVCDec Decoder object pointer.

Return value:

eAVCDecRetType

Specifies whether frames are there for output or not.
Enumeration is described in the above section. Return values are -

E_AVCD_NOERROR - Frames present for display
E_AVCD_NO_OUTPUT - No more frames for display

2.2.17 Free Decoder

This is the decoder function to release any resources used by the decoder.

Prototype:

```
eAVCDecRetType eAVCDFreeVideoDecoder (sAVCDecoderConfig *psAVCDer);
```

Arguments:

- psAVCDec Decoder object pointer.

Return value:

eAVCDecRetType

Specifies whether the decoding resources are freed.
Enumeration is described in the above section. Return values are -

E_AVCD_NO_OUTPUT - Function successful.
Other values - Error

2.2.18 Set a Buffer Manager

As section 2.2.4, 2.2.5 and 2.2.6 mentioned, a buffer manager is needed for DR. This object should be set by the application (frame work) before decoding.
The function specified below is used to set the buffer manager.

Prototype:

```
void AVCDSetBufferManager (sAVCDecoderConfig *psAVCDec,
AVCD_FrameManager* manager);
```

Arguments:

- `psAVCDec` Pointer to *`sAVCDecoderConfig`*
- `manager` a pointer to a frame buffer manager which is used to get and reject buffer.

Return value:
None

2.2.19 Set Deblock option

This function can be used to set the deblock option(Hardware or Software).

Prototype:

```
void AVCDSetDeblockOption(sAVCDecoderConfig *psAVCDec,
eAVCDDeblockOption deblockOption);
```

Arguments:

- `psAVCDec` Pointer to *`sAVCDecoderConfig`*
- `deblockOption` indicate whether us.HW or SW deblock

Return value:
None

2.2.20 Get Deblock option

This function can be used to check the deblock option(Hardware or Software) that currently used by decoder.

Prototype:

```
eAVCDDeblockOption AVCDGetDeblockOption(sAVCDecoderConfig
*psAVCDec);
```

Arguments:

- `psAVCDec` Pointer to *`sAVCDecoderConfig`*

Return value:

```
E_AVCD_SW_DEBLOCK(0)    using SW deblock
E_AVCD_HW_DEBLOCK(1)    using HW deblock
```

2.2.21 Callback type definitions

```
typedef enum
{
   E_GET_FRAME =0,
   E_REJECT_FRAME,
```

```
   E_RELEASE_FRAME
} eCallbackType;
```

Callback type	Comments
E_GET_FRAME	Set callback to get DR buffer
E_REJECT_FRAME	Set callback to reject DR buffer
E_RELEASE_FRAME	Set callback to release DR buffer

```
typedef enum
{
 E_CB_SET_OK =0,
 E_CB_SET_FAIL,
} eCallbackSetRet;
```

Callback return type	Comments
E_CB_SET_OK	Set callback successfully
E_CB_SET_FAIL	Fail to set callback

2.2.22 Set Callback function

As a additional method for DR buffer management. This object should be set by the application (frame work) before decoding.
The function specified below is used to set the buffer manager.

Prototype:

```
eCallbackSetRet
H264SetAdditionalCallbackFunction  (sAVCDecoderConfig   *psAVCDec,
eCallbackType funcType, void* cbFunc);
```

Arguments:

- psAVCDec — Pointer to *sAVCDecoderConfig*
- funcType — Indicate the function type
- cbFunc — Pointer to a callback function

Return value:

E_CB_SET_OK	set callback function successfully
E_CB_SET_FAIL	failed

2.2.23 Query decoder version

The feature to query h264 decoder version is supported through an API function, once application call the function, decoder always return the corresponding version information. The version format includes codec type, OS type, Demo type, and build time. For example:

H264D_ARM11_02.03.00_DEMO build on Jun 13 2008 13:01:27

Prototype:

```
const char * H264DCodecVersionInfo();
Arguments: void.
Return value: void.
```

3 Application Notes

3.1 Interaction between library and application

The following figure explains the various stages of the decoder, the various states it undergoes through and it's interaction with the application. This is applicable for any application that uses the H.264 library. Please note that the sample test application provided reads bitstreams that have their NALs encapsulated as specified in Annex B of AVC decoder specification [2].

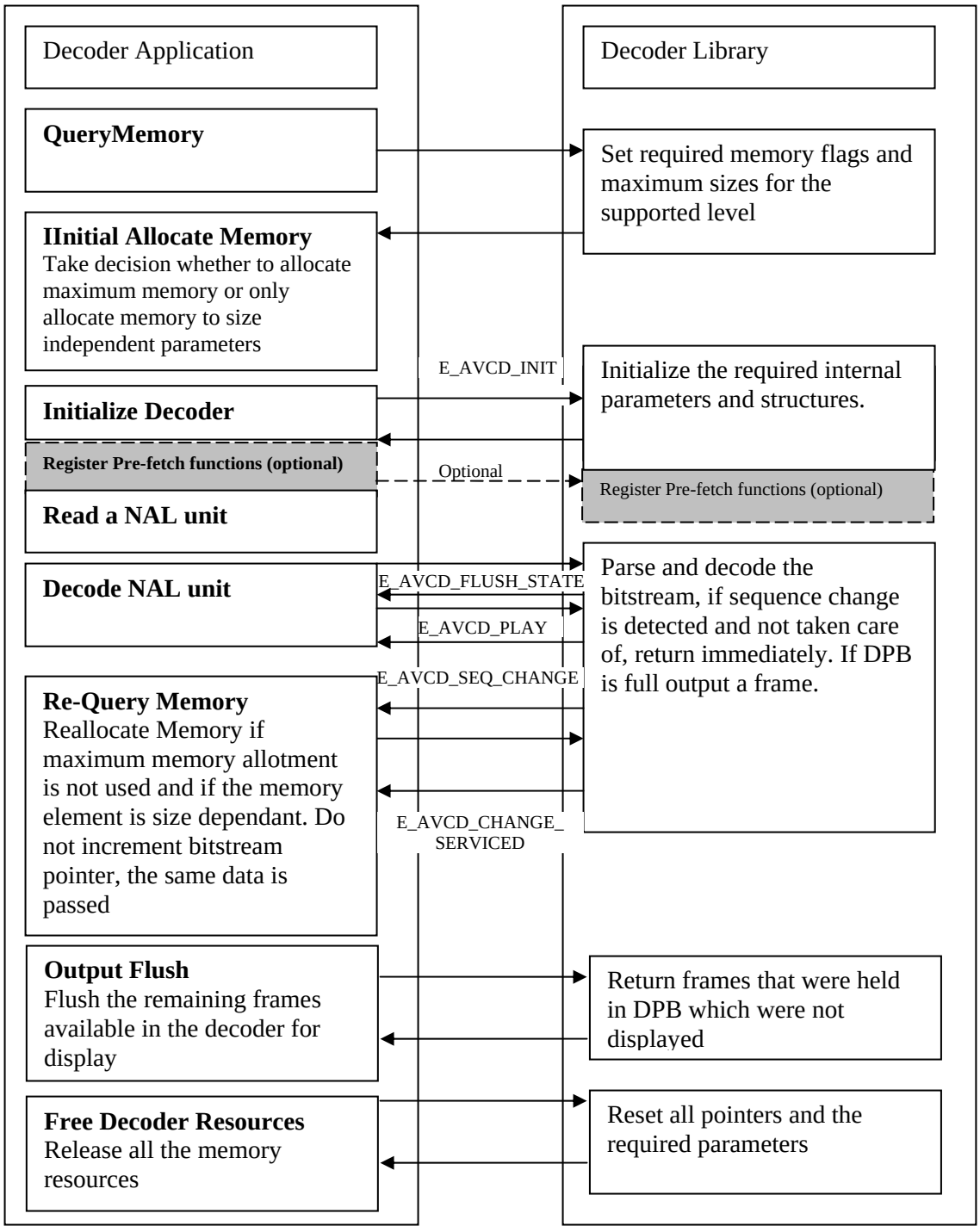

Figure 2. Interactions between Decoder Library and Application during a normal decode process

3.2 State Transitions during API calls

The following state diagrams explain the state transitions in the decoder library and the application

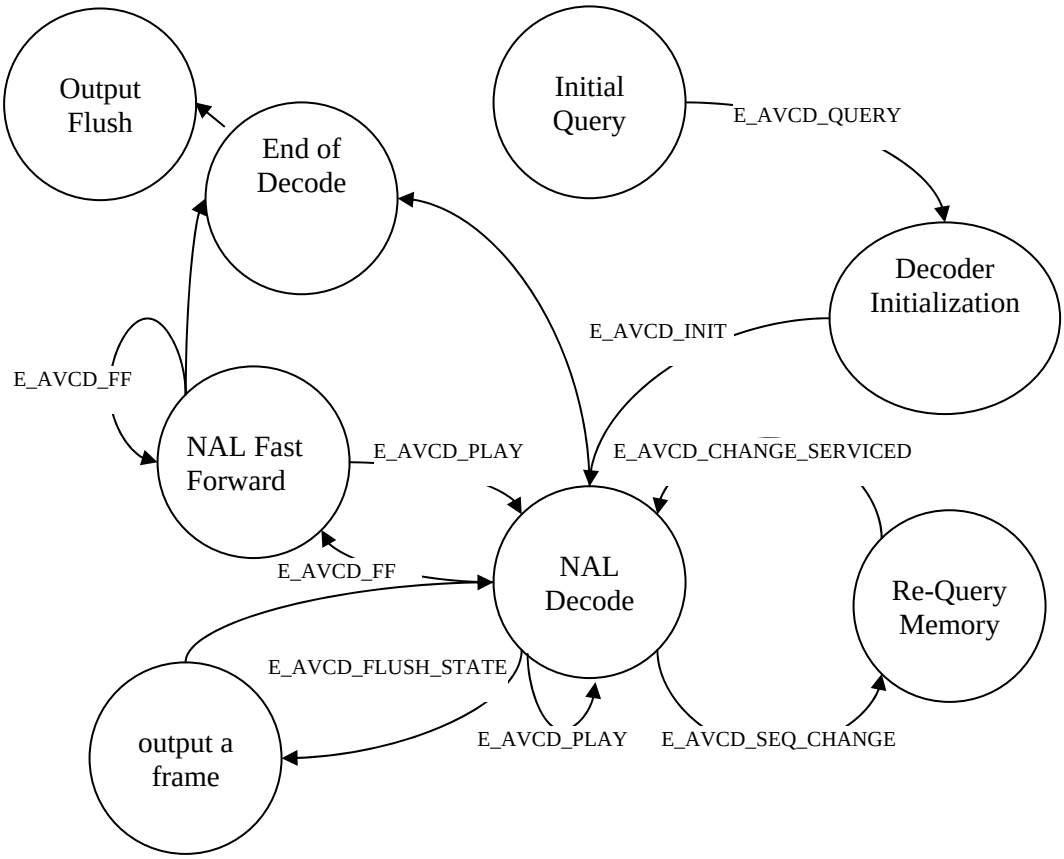

Figure 3. State Machine Indicating the State Changes when APIs are Called

3.3 Decode Flow in Applications Perspective

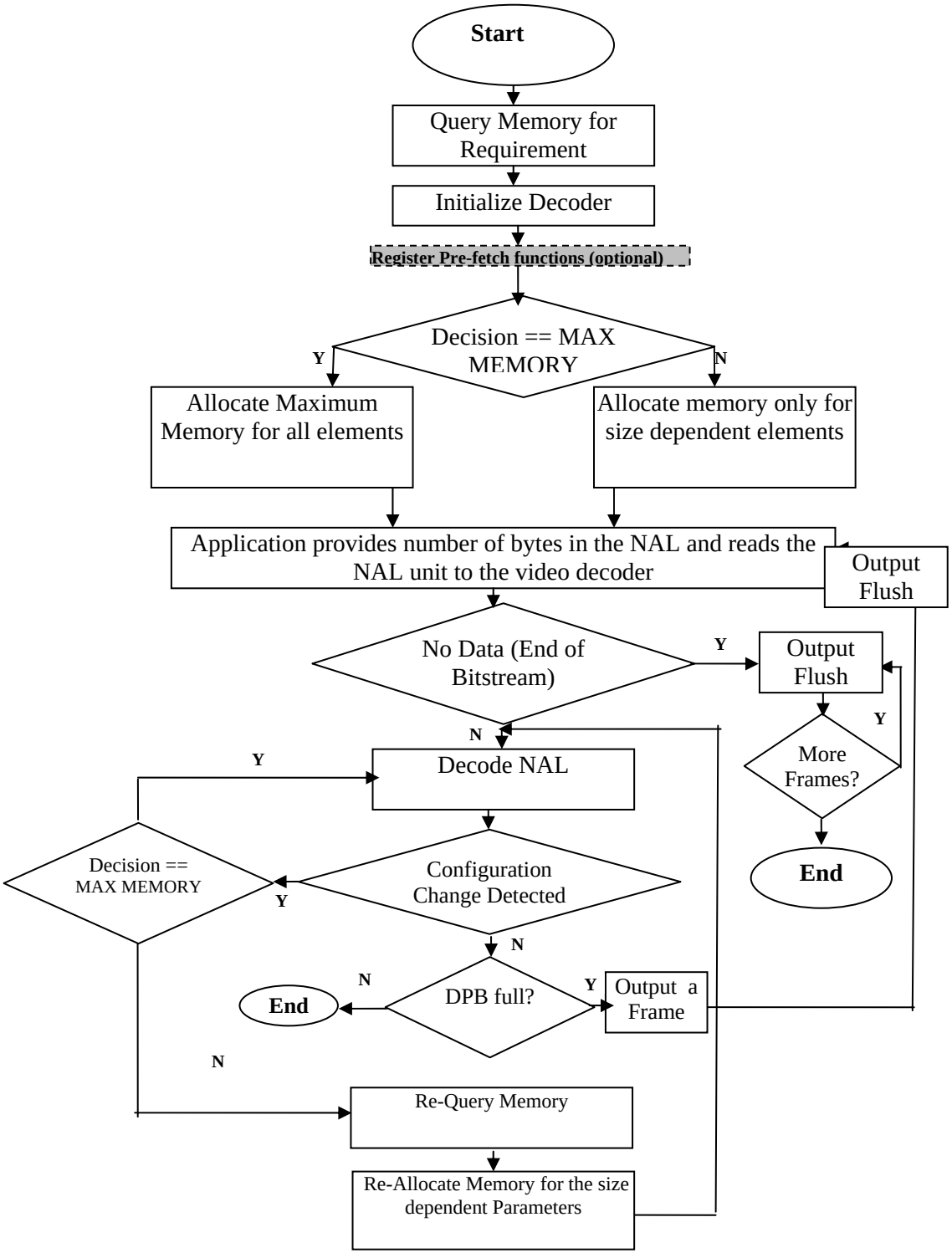

Figure 4. Application Program Flow

4 Example Lib Usage

This example shows how to use the H.264/AVC Decoder library. Before calling the decoder, we call the decoder initialization function, *eAVCDInitVideoDecoder*. This function initializes the decoder. The main decode function is called in a loop. The encoded bit stream is fed through *cbkAppBufRead* function.

Please note that the error handling is not shown properly and the code may not compile as it is. Also, few places only one component is set, while the application need to set for all of the components.

Note: This example shows the basic implementation. The same may not have updated with any change in decoder API/ structures. “decoder.c” [10] illustrates implementation of sample test application.

```
void main()
{
/******* portion of the application code ********/

/* Before calling initialization routine call QueryMem to return the size
and the type of the memory needed by the decoder, assume that the
bitBuffer contains the initial portion if bitstream and bufLen contains
the length. The application needs to register the callback function and
set the output format*/

    eStatus = eAVCDInitQueryMem (&vdec.sMemInfo);
    InitalizeAppMemory(&vdec.sMemInfo);


/* Give memory to the decoder for the size, type and alignment returned
*/
    AppAllocMemory(&vdec.sMemInfo);

/* Initialize the decoder. */
    eStatus = eAVCDInitVideoDecoder(&vdec);

/* Decode the bit stream and produce the outputs */
while (1)
{
            vdec.s32NumBytes =
            IO_GetNalUnitAnnexB(&ioPars, vdec.pvInBuffer,
                                    vdec.s32InBufferLength);
/8 IO_GetNalUnitAnnexB is implemented by the application*/

/* H264 decoder call. Returns after decoding a NAL unit*/
            eStatus = eAVCDecodeNALUnit(&vdec, ff_flag);

            if (eStatus == E_AVCD_SEQ_CHANGE)
            {
                  eStatus = eAVCDReQueryMem(&vdec);
```

```
                  Appallocmemory_1(&vdec.sMemInfo);
            }
}
                if ((vdec.sConfig.s16FrameWidth != 0) ||
                    (vdec.sConfig.s16FrameHeight != 0))
                {
                    AllocateFrameMemory(&vdec.sFrameData, &vdec.sConfig);
                    ff_flag = 0;
                }

      if ((eStatus == E_AVCD_NOERROR) || (eStatus == E_AVCD_FLUSH_STATE))
        {
        if(vdec.sFrameData.eOutputFormat != E_AVCD_420_PLANAR_PADDED)
            eAVCDGetFrame ( &vdec );
       WriteOutput(eStatus, &ioPars, vdec);
        }
}

   while(eStatus == E_AVCD_NOERROR)
   {
      eStatus=eAVCDecoderFlushAll(&vdec);
      if(eStatus == E_AVCD_NOERROR)
      {
         if(vdec.sFrameData.eOutputFormat != E_AVCD_420_PLANAR_PADDED)
             eAVCDGetFrame ( &vdec );

          WriteOutput(eStatus, &ioPars, vdec);
      }
   }
/*Free the decoder*/
    FreeDecoderMemory(&vdec.sMemInfo);
   return 0;
}


void AppAllocMemory(sAVCDMemAllocInfo *psMemPtr)
{
    int s32Count, maxNumReqs;

    maxNumReqs = psMemPtr->s32NumReqs;

    for (s32Count = 0; s32Count < maxNumReqs; s32Count++)
    {
        if (psMemPtr->asMemBlks[s32Count].s32Allocate == 1)
        {
            psMemPtr->asMemBlks[s32Count].pvBuffer = \
                MALLOC(psMemPtr->asMemBlks[s32Count].s32Size);
        }
    }
}

void AppAllocMemory_1(sAVCDMemAllocInfo *psMemPtr)
{
    int s32Count, maxNumReqs;
```

```
    maxNumReqs = psMemPtr->s32NumReqs;

    for (s32Count = 0; s32Count < maxNumReqs; s32Count++)
    {
        if (psMemPtr->asMemBlks[s32Count].s32SizeDependant == 1 &&
            psMemPtr->asMemBlks[s32Count].s32Allocate == 1)
        {
            if ((psMemPtr->asMemBlks[s32Count].pvBuffer != NULL) &&
                (psMemPtr->asMemBlks[s32Count].s32Copy == 1))
            {
                psMemPtr->asMemBlks[s32Count].pvBuffer =
                    REALLOC(psMemPtr->asMemBlks[s32Count].pvBuffer,
                            psMemPtr->asMemBlks[s32Count].s32Size);
            }
            else
            {
                 if(psMemPtr->asMemBlks[s32Count].pvBuffer)
                 {
                    free(psMemPtr->asMemBlks[s32Count].pvBuffer);
                 }
                 psMemPtr->asMemBlks[s32Count].pvBuffer = \
                 MALLOC(psMemPtr->asMemBlks[s32Count].s32Size);
            }
        }
    }
}


void AllocateFrameMemory(sAVCDYCbCrStruct *psFrame, sAVCDConfigInfo
*pConfig)
{
    int s32Xsize, s32Ysize, s32Cysize, s32Cxsize;
    if (psFrame->pu8y != NULL)
        free(psFrame->pu8y);
    if (psFrame->pu8cb != NULL)
        free(psFrame->pu8cb);
    if (psFrame->pu8cr != NULL)
        free(psFrame->pu8cr);

    // Allocate memory to the frames
    switch (psFrame->eOutputFormat)
    {
      case E_AVCD_420_PLANAR:

            psFrame->s16Xsize = (pConfig->s16FrameWidth);
            s32Xsize = psFrame->s16Xsize;
            s32Ysize = (pConfig->s16FrameHeight);
            psFrame->s16CxSize = psFrame->s16Xsize >> 1;
            s32Cysize = s32Ysize >> 1;
            s32Cxsize = psFrame->s16CxSize;
            psFrame->pu8y =(unsigned char*) MALLOC(s32Xsize * s32Ysize
                  * sizeof (char));
            psFrame->pu8cb =(unsigned char*) MALLOC(s32Cxsize * s32Cysize
```

```
                * sizeof (char));
            psFrame->pu8cr =(unsigned char*) MALLOC(s32Cxsize * s32Cysize
                * sizeof (char));
            if(psFrame->pu8y==NULL)
            {
                printf("Failed to allocate output buffer(Y)\n");
                exit(0);
            }
            if(psFrame->pu8cb==NULL)
            {
                 printf("Failed to allocate output buffer(U)\n");
                 exit(0);
            }
            if(psFrame->pu8cr==NULL)
            {
                  printf("Failed to allocate output buffer(V)\n");
                     exit(0);
            }
            break;

            case E_AVCD_420_PLANAR_PADDED:
            psFrame->s16Xsize = (pConfig->s16FrameWidth);
            s32Xsize = psFrame->s16Xsize;
            s32Ysize = (pConfig->s16FrameHeight);
            psFrame->s16CxSize = psFrame->s16Xsize >> 1;
            s32Cysize = s32Ysize >> 1;
            s32Cxsize = psFrame->s16CxSize;
            psFrame->pu8y  = NULL;
            psFrame->pu8cb = NULL;
            psFrame->pu8cr = NULL;
            break;

      case E_AVCD_422_UYVY:
            psFrame->s16Xsize = (pConfig->s16FrameWidth)*2;
            s32Xsize = psFrame->s16Xsize;
            s32Ysize = (pConfig->s16FrameHeight);
            psFrame->pu8y =(unsigned char*) malloc(s32Xsize * s32Ysize*
sizeof (char));
            if(psFrame->pu8y==NULL)
            {
                  printf("Failed to allocate output buffer(YUV)\n");
                  exit(0);
            }

                  psFrame->pu8cb = NULL;
                  psFrame->pu8cr = NULL;
            break;

   }
}


void FreeDecoderMemory(sAVCDMemAllocInfo *psMemPtr)
{
```

```
    int s32Count, maxNumReqs = psMemPtr->s32NumReqs;

    for (s32Count = 0; s32Count < maxNumReqs; s32Count++)
    {
        if (psMemPtr->asMemBlks[s32Count].pvBuffer != NULL)
        {
            free(psMemPtr->asMemBlks[s32Count].pvBuffer);
        }
    }
}

void FreeFrameMemory(sAVCDYCbCrStruct *psFrame)
{
   if ( ( psFrame->eOutputFormat == E_AVCD_422_UYVY ) || ( psFrame-
>eOutputFormat == E_AVCD_420_PLANAR ) )
   {
    if (psFrame->pu8y != NULL)
        free(psFrame->pu8y);
    if (  psFrame->eOutputFormat == E_AVCD_420_PLANAR )
    {
        if (psFrame->pu8cb != NULL)
         free(psFrame->pu8cb);
        if (psFrame->pu8cr != NULL)
         free(psFrame->pu8cr);
    }
   }
}
```

5 Debug Logs

The purpose of providing the developer with useful information that not only could be used during debugging the library but also to understand the dataflow in H.264 video encoder. These logs are classified into four categories namely DebugLog1, DebugLog2, DebugLog3 and DebugLog4 according to the level of hierarchy that is addressed. The hierarchies addressed by the logs are shown in Table 1 below

Table 1. Hierarchical levels used for Debug Logs

Type of Log	Functionality Addressed	Module ID	Examples
DebugLog1 (DEBUG_1)	Sequence, Picture and API Parameters	INTERFACE, SEQUENCE, PICTURE, BITSTREAM	Sequence Parameter, Picture Parameters, type of NAL etc.
DebugLog2 (DEBUG_2)	VOP Level Parameters	PSLICE, ISLICE, ERRORC, MEMMGT, BITSTREAM, DEBLOCK	Frame type, Macroblocks types in a frame, Quantization parameters in a frame etc.
DebugLog3 (DEBUG_3)	Macroblock level parameters	PSLICE, ISLICE, BITSTREAM,	Macroblock type, coded bit pattern, etc.
DebugLog4 (DEBUG_4)	Block level Parameters	PSLICE, ISLICE, BITSTREAM, MISC, BITSTREAM	Paths relating to block coefficients, runs, levels etc.

Enumerations are defined and provided in a separate header file for the user to view the various types of information that could be logged using the given debug APIs. These enumerations specify both the module level and the message type that are to be logged. Using the debug logs judiciously inside the implementation, messages are generated in an ordered form. The types of message types for each of the module used in the current version of the encoder are as shown in Table 2.

Table 2. Message Types in each of the Hierarchies

Message Type	Comments
Message Identity (MSGIDS_START)	States the start of message logging for a given function
Function Information (FN_INFO)	Generic information of the function, for e.g., function entry/exit, information on the I/O etc.
General Information (GENERAL_INFO)	Information and values of various processing steps and parameters that a function executes.
End of Message Identity (MSGIDS_END)	Flags the end of the messages in a function.

5.1 Logger Functions

The H264 decoder does not have any logger function to the log the text and data information by itself. The logger function must be provided by the application to the library through function interface *AVCD_Register_Debug_Funcptr*. The logger functions are passed as arguments to this function. This function must be called before calling any other H264 decoder interface functions. If this function is not called by the application, then the H264 decoder considers the Debug Level as inactive.

The function prototype for *AVCD_Register_Debug_Funcptr* is as below.

C prototype:

```
void AVCD_Register_Debug_Funcptr ( \
int (*text_debug_ptr)(short int msgid,char *fmt,...), \
int (*data_debug_ptr)(short int msgid,void *ptr,int size));
```

Arguments:

- text_debug_ptr - Text Debug Log function pointer.
- data_debug_ptr - Data Debug Log function pointer.

Return Value:

None

There are two logger functions which are used to log the text and data respectively.
1. *DebugLogText (short int msgid, char *fmt, ...)*

Where, msgid is used to identify the module and the action required.
fmt is a character string with the format same as "*printf()*" in 'C'.

2. *DebugLogData (short int msgid, int size, void *ptr)*
where *size* - size of data in bytes.
ptr - pointer of data to be logged.

In addition another added capability of the provided logs is the specific debug API for providing function entry/exit points in the logged data. The current version of implementation treats this very similar to that of the debug log text. This is because of the fact that API is primarily used just to denote whether functions that are entered are executed in total without encountering any problems.

6 Appendix

6.1 Pre-fetch callback function implementation

Here are example implementations of Pre-fetch functions that are mentioned in section 2.1.7

```
#define MAX_NAL_LEN  2100000

// Application buffers 2 NAL
UCHAR NALBuffer[2][MAX_NAL_LEN];
int current_nal_index = -1;//Which NALBuffer is used
int NALBufferBytes[2];//Length of NALBuffer
int NALDeStuffed[2];//NAL destuffed flag

int PrefetchNAL(UCHAR **pbuf, int *len)
{
    int prefetch_nal_index;
    if(current_nal_index == 0)
        prefetch_nal_index = 1;
    else
        prefetch_nal_index = 0;

    *len = NALBufferBytes[prefetch_nal_index];
    *pbuf = NALBuffer[prefetch_nal_index];

    if(*len == 0)
        return -1;
    else
        return 0;//ok
}
void SetPrefetchNALLength(int len_after_destuff)
{
    int prefetch_nal_index;
    if(current_nal_index == 0)
        prefetch_nal_index = 1;
    else
        prefetch_nal_index = 0;

    NALBufferBytes[prefetch_nal_index] = len_after_destuff;
    NALDeStuffed[prefetch_nal_index] = 1;
}
void init_NAL_buffers() /*which is invoked by application to initialize the NAL
buffer*/
{
    current_nal_index = 0;
    NALBufferBytes[0] = NALBufferBytes[1] = 0;
    NALDeStuffed[0] = NALDeStuffed[1] = 0;
}

void read_nal(int index) /*which is invoked by application to prepare NAL data*/
{
    NALBufferBytes[index] = IO_GetNalUnitAnnexB(&ioPars, NALBuffer[index],
MAX_NAL_LEN);
    NALDeStuffed[index] = 0;
}
```