Application Report
SPRAB88–December 2009

Understanding H.264 Decoder Buffer Mechanism for TMS320DM365

Krishnarao Penumutchu and Yashwant Dutt

ABSTRACT

With XDM1.0, the decoder does not have separate reference and display buffers. The reference buffers of the decoder are used for display as well. This helps in reducing the overall DDR utilization and DDR b/w. However, this new arrangement changes the overall buffer management of the application. Now the application needs to manage the display and reference buffers using the new XDM 1.0 APIs. This application report provides information on how to manage the decoder buffer with respect to H.264 from the application writer's perspective. The first section of the document gives an overview of the change, the second section describes how the buffers are initialized, ownership of the buffers and how the application gets the ownership of the buffers, which is passed to codec. The third section describes the sample buffer manager. Finally, the sample test application is described. This document is not only applicable to DM365 platform, but also similar other platforms and codecs.

Contents

List of Figures

1 Introduction

H.264 is a popular video coding algorithm enabling high-quality multimedia services on limited bandwidth network. DM365 is a digital multi-media system on-chip (SoC) primarily used for video security, video conferencing and other applications. DM365 has an ARM926 host processor, two co-processors and various peripherals. DM365 also has a video front-end hardware block that enables seamless interface to most additional external devices required for video applications.

Buffer management is the key element for proper functionality of the codec and the application. With XDM 1.0, the decoder does not ask for the frame buffer at the time of `alg_create()`. It uses buffer from `XDM1_BufDesc *outBufs`, which it gets during each decode process call. Therefore, there is no distinction between the Decoder Picture Buffer (DPB) and display buffers. The application framework just needs to ensure that it does not overwrite the buffers, which are locked by codec. To perform this application requires a buffer manager that maintains the status of all the buffers, i.e., which are free and locked by codec. At the end of each decode process call, the application checks for any buffers unlocked by codec. On finding free buffers, the application invokes the buffer manager and then changes the status of the buffers.

2 Buffer Management by Application

The application needs to provide buffers to the codec module for filling the reference frames and output buffers for display. In every process call to the codec module, application provides one free buffer identified by a unique buffer ID. Buffer IDs are positive non-zero numbers. Initially, the application does

not know the exact size of the picture width and height. Hence, the application initializes the buffer manager with sufficient buffer size for output and reference frames. The buffer size value specified to initialize the buffer manager is dependent on the level supported. At the end of first decode call, application calls GETBUFINFO control call to know the exact picture width and height. Once the application knows the exact picture width and height then it calls buffer manager for re-initialization. This re-initialization is necessary.

There is no distinction between codec and application buffers. The application framework needs to ensure that it does not overwrite the buffers that are locked by the codec.

```
BUFFMGR_Init();  /* Initialize the buffer manager with the size supported by the level */
H264VDEC_create();
H264VDEC_control(XDM_GETBUFINFO);  /*Returns default picture width and height creation
                                     parameters */
do
{
  BUFFMGR_GetFreeBuffer();  /*Get a handle to the available buffer from buffer manager */
  H264VDEC_decode()         /*Call the decode API */
  if(frames decoded == 1)
  {
    H264VDEC_control(XDM_GETBUFINFO);   /* Updates the memory required as per the size
                                           parsed in stream header */
    BUFFMGR_ReInit();       /*Reinitialize the buffer manager based on picture width and
height*/
  }
  BUFFMGR_ReleaseBuffer     /*if codec releases any buffers update them in buffer manager */
}
While(all frames)
ALG_delete ();             /*Delete the Codec instance object handle once the decoding over */
BUFFMGR_DeInit();          /*Releases all memory allocated by buffer manager*/
```

The frame pointer provided by the application and that returned by the algorithm may be different. `BufferID (InputID/outputID)` provides the unique ID to keep record of the buffer given to and released by the algorithm.

The application framework can efficiently manage frame buffers by keeping a pool of free buffers from which it gives the free buffer to the decoder on request.

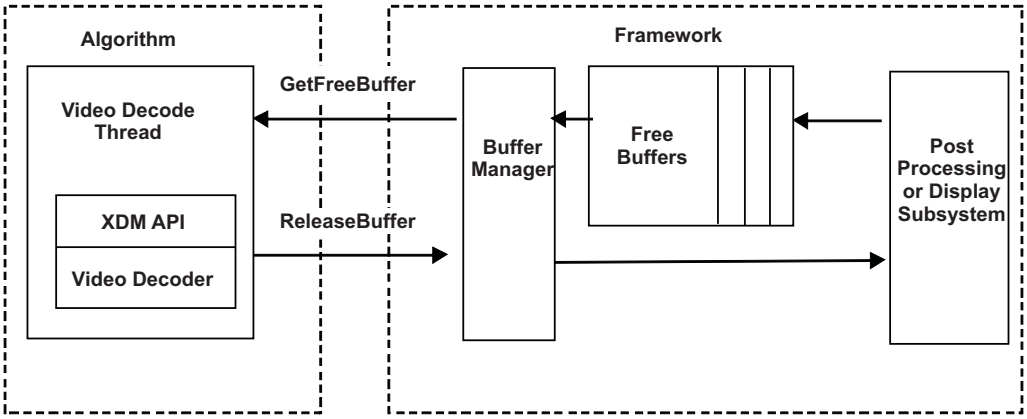

Figure 1. Interaction of Frame Buffers Between Application and Framework

2.1 Ownership of Buffers

Before every process call, the application provides a free buffer to the codec through buffer manager, and the buffer manager assigns the status of the buffer as unavailable. Buffers, once accessed by the application to the codec module, are by default owned by the codec module until explicitly released by it. Codec layer is required to use this buffer for filling in currently decoded frames. Buffers so obtained by the application can be retained by the codec layer as long as it desires.

2.2 Displaying Output Buffers

Successfully decoded frames can be displayed (after satisfying display delay, display order) by initializing the first elements of the `outArgs.displayBufs[ ]` structure array with suitable buffer pointers and frame properties. Corresponding entry in the output array `OutArgs.outputID[ ]` provides the buffer ID of the display buffer. There can be multiple display buffers in return to a single process call; all these display buffers are indicated by non-zero positive buffer Id entries in `outArgs.outputID[ ]` array. The number of display buffers present is given by the number of non-zero positive buffer Id entries in `outArgs.outputID[ ]` array. Already displayed buffers are still under codec ownership until they are explicitly released by the codec.

2.3 Releasing Displayed Buffers

Already displayed and no more internally referenced buffers can be released by the codec by populating corresponding buffer Ids in the `outArgs.freeBufID[ ]` array. There can be multiple buffers freed at the same time in one single process call. The number of currently released free buffers is equal to the number of non-zero positive buffer id entries in the `outArgs.freeBufID[ ]` array. The application gets the ownership of the released buffers and the buffer manager assigns the status of these buffers as available.

2.4 Dependency of Buffer Manager on Display Delay Parameter

In cases where the frame order in encoded bitstream is different than the display order, some display delay is necessary for proper display.

Application framework provides the *displayDelay* parameter to the codec through `IH264VDEC_Params`. Display delay parameter must always be less than the DPB limit. If the frame work provides display delay more than DPB limit, then the codec assigns the display delay parameter equal to DPB limit. If the display delay is greater than zero, then the codec stores the decoded frame in DPB. In the DPB, if the number of frames to display is greater than display delay, then the codec displays a frame that has low POC and the same is updated in `outArgs.displayBufs[ ]`. If the displayed frame is unreferenced, then it removes the frame from DPB and the same is updated in `outArgs.freeBufID[ ]`. So based on the DPB operation, clearly there is a dependency between the buffer manager, display delay and max_ref_frame defined by the level limit. The buffer manager must contain (*X* + *1*) number no of buffers where *X* is max of num_ref_frame or display delay. After first frame decode application calls, the buffer manager for re-initialization. In re-initialization, buffer manager accommodates (*X* + *1*) number no of buffers. Some examples are mentioned below that describe how the codec locks and unlocks the buffers based on display delay and the DPB size.

Example 1

No of reference frames = 2, Display delay = 0

Input ID	1	2	3	4	5	6	7
Output ID	1	2	3	4	5	6	7
Freebuf ID			1	2	3	4	5
Frame No	0	1	2	3	4	5	6

Example 2

No of reference frames = 3, Display delay = 2

Input ID	1	2	3	4	5	6	7
Output ID			1	2	3	4	5
Freebuf ID				1	2	3	4
Frame No	0	1	2	3	4	5	6

Example 3

No of reference frames = 3, Display delay = 2 and Input ID = 6 is an IDR picture

Input ID	1	2	3	4	5	6	7	8	9	10
Output ID			1	2	3	4, 5		6	7	8
Freebuf ID				1	2	3, 4, 5			6	7
Frame No	0	1	2	3	4	0	1	2	3	4

If the application has constraints on the amount of memory it can allocate for DPB, a customized solution can be created to reduce memory footprint depending on the nature of input bitstream. If the application scenario is such that the max_num_ref frames will not be more that 2 and the display delay is set to 0, then the DPB can be managed by 3 frames. Using Example 1, the decoder starts freeing frame from frame 2 onwards. This free frame can be used by the application to once again feed the codec.

3 Sample Buffer Manager

The functions of the buffer manager in application can be implemented in various ways. The set of functions below illustrate a sample-use case. The code snippets in the document are for reference only.

BUFFMGR_Init()— The `BUFFMGR_Init` function is called by the test application to initialize the global buffer element array to default and to allocate the required number of memory data for reference and output buffers. The maximum required DPB size is defined by the supported profile and level.

BUFFMGR_ReInit()— The `BUFFMGR_Init` function allocates global luma and chroma buffers and allocates entire space to first element. This element is used in the first frame decode. After the first frame decode luma and chroma buffer requirements are obtained, the global luma and chroma buffers are re-initialized to other elements in the buffer array.

BUFFMGR_GetFreeBuffer()— The `BUFFMGR_GetFreeBuffer` function searches for a free buffer in global buffer array and returns the address of that element. If none of the elements are free, then it returns NULL.

BUFFMGR_ReleaseBuffer()— The `BUFFMGR_ReleaseBuffer` function takes an array of buffer-ids that are released by the codec. “0" is not a valid buffer Id, therefore, this function keeps moving until it encounters a buffer Id as zero or it hits the MAX_BUFF_ELEMENTS.

BUFFMGR_DeInit()— The `BUFFMGR_DeInit` function releases all memory allocated by buffer manager.

3.1 H264Decoder Buffer Manager Sample Code

```
/*@func BUFFMGR_Init()
*@brief TI's implementation of buffer manager initialization module
*The BUFFMGR_Init function is called by the test application to initialise the global *buffer
element array to default and to allocate required number of memory data for *reference and
output buffers.The maximum required dpb size is defined by the *supported profile & level.
*@param  totBufSize: Total buffer size to be allocated
*@return Success(0)/failure(-1) in allocating and initialising
*/

XDAS_Int32 BUFFMGR_Init(XDAS_Int32 totBufSize)
{
    XDAS_UInt32 tmpCnt;
    /* total buffer size allocatable is divided into three parts one part goes
     * to luma buffers and the other two parts go to luma buffers */
    chromaGlobalBufferSize = (totBufSize/3);
    lumaGlobalBufferSize = (chromaGlobalBufferSize*2);

    if(lumaGlobalBufferHandle == NULL)
    {
        /* Allocate the global buffers */
        lumaGlobalBufferHandle = malloc(lumaGlobalBufferSize);
        if(lumaGlobalBufferHandle == NULL)
        {
            return -1;
        }
        chromaGlobalBufferHandle = malloc(chromaGlobalBufferSize);
        if(chromaGlobalBufferHandle == NULL)
        {
            free(lumaGlobalBufferHandle);
            return -1;
        }
    }
       memset(lumaGlobalBufferHandle, 0x66, lumaGlobalBufferSize);
       memset(chromaGlobalBufferHandle, 0x66, chromaGlobalBufferSize);
    /* Initialise the elements in the global buffer array */
    for(tmpCnt = 0; tmpCnt < MAX_BUFF_ELEMENTS; tmpCnt++)
    {
        buffArray[tmpCnt].bufId = tmpCnt+1;
        buffArray[tmpCnt].bufStatus = BUFFMGR_BUFFER_FREE;
        buffArray[tmpCnt].bufSize[1] = 0;
        buffArray[tmpCnt].bufSize[0] = 0;
        buffArray[tmpCnt].buf[1] = NULL;
        buffArray[tmpCnt].buf[0] = NULL;
    }

    /* Initialise the entire buffer space to first frame and re-modify buffer sizes
     * as per the picture frame sizes after first frame decode */
     buffArray[0].buf[1] = chromaGlobalBufferHandle;
     buffArray[0].bufSize[1] = chromaGlobalBufferSize;
     buffArray[0].buf[0] = lumaGlobalBufferHandle;
     buffArray[0].bufSize[0] = lumaGlobalBufferSize;
     return 0;
}
```

```
*@func BUFFMGR_ReInit()
*@brief TI's implementation of buffer manager re-initialization module
* The BUFFMGR_Init function allocates global luma and chroma buffers and allocates
* entire space to first element. This element will be used in first frame decode.
* Once the picture's height and width are known luma and chroma buffer requirements
* are obtained then the global luma and chroma buffers are re-initialised to other
* elements in the buffer arary.
*@param  lumaOneFrameBufSize: Buffer size for one luma frame in bytes
*@param  chromaOneFrameBufSize: Buffer size for one chroma frame in bytes
*@return Success(0)/failure(-1) in allocating and initialising
*/
XDAS_Int32 BUFFMGR_ReInit(XDAS_Int32 lumaOneFrameBufSize,
                          XDAS_Int32 chromaOneFrameBufSize)
{
    XDAS_Int32 tmpLum, tmpChrm, tmpCnt;
    XDAS_UInt8 *tmpLumaBuf, *tmpChrmBuf;

    /* check if the requested sizes exceed allocated buffer sizes */
    if((lumaOneFrameBufSize > lumaGlobalBufferSize) ||
       (chromaOneFrameBufSize > chromaGlobalBufferSize))
    {
        return -1;
    }

    tmpLum = lumaGlobalBufferSize;
    tmpChrm = chromaGlobalBufferSize;
    tmpLumaBuf = lumaGlobalBufferHandle;
    tmpChrmBuf = chromaGlobalBufferHandle;
    /* now re-allocate buffer sizes for each element based on the
     * per frame buffer requirements */
    for(tmpCnt = 0;
        (tmpCnt < MAX_BUFF_ELEMENTS) &&
        ((tmpLum - lumaOneFrameBufSize) >= 0) &&
        ((tmpChrm - chromaOneFrameBufSize) >= 0);
        tmpCnt++)
    {
        buffArray[tmpCnt].buf[0] = tmpLumaBuf;
        buffArray[tmpCnt].buf[1] = tmpChrmBuf;

        buffArray[tmpCnt].bufSize[1] = chromaOneFrameBufSize;
        buffArray[tmpCnt].bufSize[0] = lumaOneFrameBufSize;
        /* update the local variables for next iteration */
        tmpLum -= lumaOneFrameBufSize;
        tmpChrm -= chromaOneFrameBufSize;
        tmpLumaBuf += lumaOneFrameBufSize;
        tmpChrmBuf += chromaOneFrameBufSize;
    }
    return 0;
}
```

```
*@func BUFFMGR_GetFreeBuffer()
*@brief TI's implementation of buffer manager re-initialization module
*The BUFFMGR_GetFreeBuffer function searches for a free buffer in the global buffer
*array and returns the address of that element. Incase if none of the elements are
*free then it returns NULL
*@return Valid buffer element address or NULL incase if no buffers are empty
*/
BUFFMGR_buffEleHandle BUFFMGR_GetFreeBuffer()
{
    XDAS_UInt32 tmpCnt;

   for(tmpCnt = 0;(tmpCnt < MAX_BUFF_ELEMENTS) && (buffArray[tmpCnt].buf[0] != NULL);
       tmpCnt++)
    {
        /* Check for first empty buffer in the array and return its address */
        if(buffArray[tmpCnt].bufStatus == BUFFMGR_BUFFER_FREE)
        {
            buffArray[tmpCnt].bufStatus = BUFFMGR_BUFFER_USED;
            return (&buffArray[tmpCnt]);
        }
    }
    /* Incase if no elements in the array are free then return NULL */
    return NULL;
}
```

```
*@func BUFFMGR_DeInit()
*@brief  TI's implementation of buffer manager re-initialization module
*The BUFFMGR_DeInit function releases all memory allocated by buffer manager.
*@return None
*/
void BUFFMGR_DeInit()
{
    if(lumaGlobalBufferHandle)
        free(lumaGlobalBufferHandle);
    lumaGlobalBufferHandle = NULL;
    if(chromaGlobalBufferHandle)
        free(chromaGlobalBufferHandle);
    chromaGlobalBufferHandle = NULL;
    return;
}
```

```
*@func BUFFMGR_ReleaseBuffer(buffId)
*@brief  TI's implementation of buffer manager re-initialization module
*The BUFFMGR_ReleaseBuffer function takes an array of buffer-ids which are
*released by the codec. "0" is not a valid buffer Id hence this function keeps
*moving until it encounters a buffer Id as zero or it hits the MAX_BUFF_ELEMENTS
*@return None
*/
void BUFFMGR_ReleaseBuffer(XDAS_UInt32 bufffId[])
{
    XDAS_UInt32 tmpCnt, tmpId;
    for(tmpCnt = 0;
        (tmpCnt < MAX_BUFF_ELEMENTS);
        tmpCnt++)
    {
        tmpId = bufffId[tmpCnt];
        /* Check if the buffer Id = 0 condition has reached. zero is not a
         valid buffer Id hence that value is used to identify the end of
         buffer array */
        if(tmpId == 0)
        {
           break;
        }
      /* convert the buffer-Id to its corresponding index in the global array */
        tmpId--;

        if(tmpId >= MAX_BUFF_ELEMENTS)
        {
         /*Inidcates an invalid buffer Id passed - this buffer Id can be
           ignored!! alternatively we can break here.*/
         printf("Trying to release a buffer using an invalid bufferId %d \
                   ignoring..\n", tmpId+1);
         continue;
        }
        /* Set the status of the buffer to FREE */
        buffArray[tmpId].bufStatus = BUFFMGR_BUFFER_FREE;
    }
    return;
}
```

4 Flowchart for Sample Test Application

1. Call `BUFFMGR_Init` (with maximum DPB size)
2. Fill-in the decoder specific params in the extended form of the `IVIDDEC1_Params`.
   (a) These params can be the maximum profile and level to be supported.
   (b) Display delay to be supported.
   (c) Display order or decode order to be supported and,
   (d) Any other user-defined codec specific params.
3. Call `ALG_Create` function and create the decoder instance.
4. Get maximum and minimum buffer requirements using the control function.
5. Call the `BUFFMGR_GetFreeBuffer` function to obtain the first free buffer from the buffer manager.
   If the above call results in NULL, the buffers have run out. However, this condition should not occur.
6. Call process function of the decoder instance with:
   (a) `inputBufDesc` populated with input encoded stream
   (b) `outputBufDesc` with luma and chroma buffer pointers as contained in the `buffElement` structure obtained by the previous call of `the BUFFMGR_GetFreeBuffer` function of the buffer manager module.
   (c) `inArgs.inputId` equal to the unique buffer id of the buffer obtained from the buffer manager using the previous `BUFFMGR_GetFreeBuffer`
   (d) `inArgs.numBytes` populated with the size of the input encoded buffer stream.

7. If the process call returns an error, then enters error handling section:
   (a) If process call returns successfully, check whether there are any non-zero positive buffer Id entries in the `outArgs.outputID[ ]` array. Corresponding entries in the `outArgs.displayBufs[]` provide pointers to the actual display buffers with picture height, width and pitch as set in frameHeight , frameWidth and framePitch elements of the displayBufs element.
8. Call control function with control command XDM_GETBUFINFO to obtain the actual width and height of the input bitstream, if the above process call was the first process call.
9. Call `BUFFMGR_ReInit` function to re-initialize the buffer manager with the actual picture width and height. This re-initialization is necessary because the buffer manager would otherwise be initialized to provide only one buffer with the entire DPB size in the `BUFFMGR_Init` function and the `outArgs.freeBufID[ ]` array would contain a list of buffer Ids that has been released by the codec component in this process call.
10. Release these buffers back to the buffer manager by calling the function:
    `BUFFMGR_ReleaseBuffer((XDAS_UInt32 *)outArgs.freeBufID);`
11. Execute steps 5 to 11 repeatedly until the loop-break condition is reached. The loop-break condition can be any of the following, whichever occurs first:
    (a) End of encoded bit-stream reached
    (b) User given number of frames is decoded
    (c) Buffer manager has exhausted with all buffers
    (d) Process call returns with `XDM_FAIL` condition

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265