HEVC/H.265 Main Profile Decoder on ARM

User's Guide

Literature Number: SPRUGH8
May 2015

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive & Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications & Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers & Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energyapps
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video & Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) ARM HEVC/H.265 Decoder implementation on ARM platform. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

Intended Audience

This document is intended for system engineers who want to integrate TI's codecs with other software to build a multimedia system based on ARM platform and Microsoft Visual Studio.

This document assumes that you are fluent in the C language, working knowledge of Linux.

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction**, provides an overview of the codec and lists its supported features.
- **Chapter 2 - Installation Overview**, describes how to install, build and run the codec.
- **Chapter 3 - Sample Usage**, describes the sample usage of the codec.
- **Chapter 4 - API Reference**, describes the data structures and interface functions used in the codec.
- **Chapter 5 - Frequently Asked Questions,** provides answers to few frequently asked questions related to using this decoder.
- **Appendix A – Meta Data Support,** provides information on writing out the parsed SEI, VUI data into application provided buffers.
- **Appendix B – Low Delay Interface,** provides information on using Call back API function usage for low latency applications.

Related Documentation

You can use the following documents to supplement this user guide:

- High Efficiency Video Coding (HEVC), Recommendation ITU-T H.265 (10/2014), ISO/IEC 23008-2.

Abbreviations

The following abbreviations are used in this document.

Table 0-1 List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
ARM	Advanced RISC Machine
CABAC	Context-based Adaptive Binary Arithmetic Coding
CB	Coding Block
CRA	Clean Random Access
CTB	Coding Tree Block
CTU	Coding Tree Unit
CU	Coding Unit
DPB	Decoded Picture Buffer
DSP	Digital Signal Processor
EVM	Evaluation Module
GOP	Group Of Pictures
HEVC	High Efficiency Video Coding
ISO	International Organization for Standardization
ITU	International Telecommunication Union
JCT-VC	Joint Collaborative Team on Video Coding
MPEG	Moving Picture Experts Group
PPS	Picture Parameter Set
PU	Prediction Unit

Abbreviation	Description
SAO	Sample Adaptive Offset
SEI	Supplemental Enhancement Information
SPS	Sequence Parameter Set
TB	Transform Block
TU	Transform Unit
VPS	Video Parameter Set
VUI	Video Usability Information

Text Conventions

The following conventions are used in this document:

- Text inside back-quotes (‘‘) represents pseudo-code.
- Program source code, function and macro names, parameters, and command line commands are shown in a `mono-spaced` font.

Product Support

When contacting TI for support on this codec, quote the product name (ARM HEVC/H.265 Decoder) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

This page is intentionally left blank

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides an overview of TI's implementation of the HEVC/H.265 Decoder on ARM and its supported features.

Topic **Page**

1.1 Overview of HEVC/H.265 Decoder

High Efficiency Video Coding (HEVC/H.265) is a video compression standard, a successor to H.264/MPEG-4 AVC (Advanced Video Coding), which was jointly developed by the ISO/IEC Moving Picture Experts Group (MPEG) and ITU-T Video Coding Experts Group (VCEG). The main goal of the HEVC standardization effort is to enable significantly improved compression performance relative to existing standards in the range of 50% bit rate reduction for equal perceptual video quality. The new advancements and greater compression ratios available at a very low bit rate has made devices ranging from mobile and consumer electronics to set-top boxes and digital terrestrial broadcasting to use the H265 standard.

Figure 1-1 depicts the working of the H.265 Decoder algorithm.

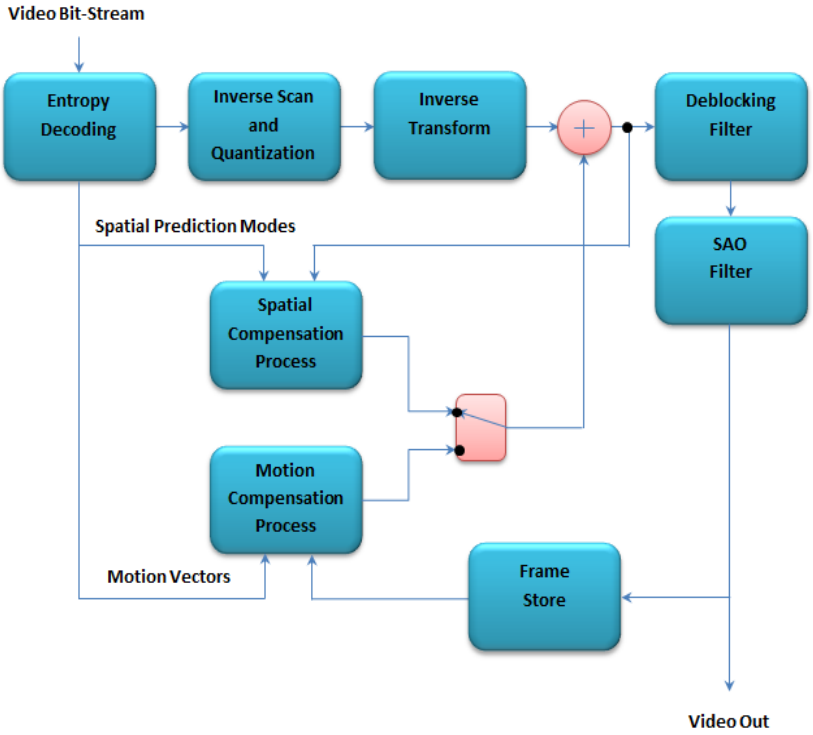

Figure 1-1 Flow diagram of the HEVC/H.265 Decoder

The video coding layer of H265/HEVC employs the same “hybrid” approach (inter/intra prediction and 2D transform coding) used in all video compression standards. *Figure 1-1* depicts the block diagram of HEVC video decoder.

In HEVC, coding efficiency is significantly improved by utilizing macroblock structures with sizes larger than 16x16 pixels, especially at high resolutions. This is due to the ability of large motion and transform blocks to more efficiently exploit the increased spatial correlation that occurs at such high resolutions. At high resolutions there are more likely to be large homogenous areas that can be efficiently represented by larger block sizes.

Each picture is splitted into Coding Tree Units (CTUs). The CTU consists of a luma coding tree block (CTB) and the corresponding chroma CTBs. The size LxL of a luma CTB can vary as L = 16, 32, or 64 samples, with the larger sizes typically enabling better compression. HEVC supports partitioning of the CTBs into smaller blocks using a tree structure and quadtree-like signaling.

The quadtree syntax of the CTU specifies the size and positions of its luma and chroma coding blocks (CBs). The splitting of a CTU into luma and chroma CBs is signaled jointly. One luma CB and ordinarily two chroma CBs, together with associated syntax, form a Coding Unit (CU). A CTB may contain only one CU or may be split to form multiple CUs, and each CU has an associated partitioning into prediction units (PUs) and a tree of transform units (TUs) as shown in *Figure 1-2*.

The first picture of a video sequence (and the first picture at each “clean” random access point into a video sequence) is coded using only intra-picture prediction (which uses some prediction of data spatially from region-to-region within the same picture but has no dependence on other pictures). For all remaining pictures of a sequence or between random access points, inter-picture temporally-predictive coding modes are typically used for most blocks.

In Spatial compensation process, decoded boundary pixels of adjacent blocks are used as reference data for spatial prediction. Intra prediction supports 33 directional modes (compared to 7 such modes in H.264/MPEG-4 AVC), plus planar (surface fitting) and DC (flat) prediction modes. Prediction modes are derived from the bit-stream and corresponding prediction block is formed.

In Motion compensation process, prediction is formed using motion vector information signaled in the bitstream and advanced motion vector prediction derived from MVs of neighbouring prediction blocks (PBs). Quarter-sample precision is used for the MVs, and 7-tap or 8-tap filters are used for interpolation of fractional-sample positions. Similar to H.264/MPEG-4 AVC, multiple reference pictures are used. For each PB, either one or two motion vectors can be transmitted, resulting either in uni-predictive or bi-predictive coding, respectively. Scaling and offset operation may be applied to the prediction signal(s) in a manner known as weighted prediction.

The residual is decoded after applying inverse quantization and inverse transform (using block transforms) on Quantized transform coefficients which are decoded from bitstream. A transform unit (TU) tree structure has its root at the CU level. The luma CB residual may be identical to the luma transform block (TB) or may be further split into smaller luma TBs. The same applies to the chroma TBs. Integer basis functions similar to those of a inverse discrete cosine transform (IDCT) are defined for the square TB sizes 4×4, 8×8, 16×16, and 32×32. For the 4×4 transform of intra-picture luma prediction residuals, an integer transform derived from a form of inverse discrete sine transform (IDST) is alternatively specified.

The residual data is added to prediction data to form the reconstructed pixels.

To avoid blocking artifacts a de-blocking filter is applied on the reconstructed pixels. After de-blocking process, Sample Adaptive offset (SAO) filter is applied on pixels. It is a non-linear amplitude mapping process and its goal is to better reconstruct the original amplitudes. Filtered frame is stored in a frame-store called DPB (decoded picture buffer) and used as a reference for next frame decoding.

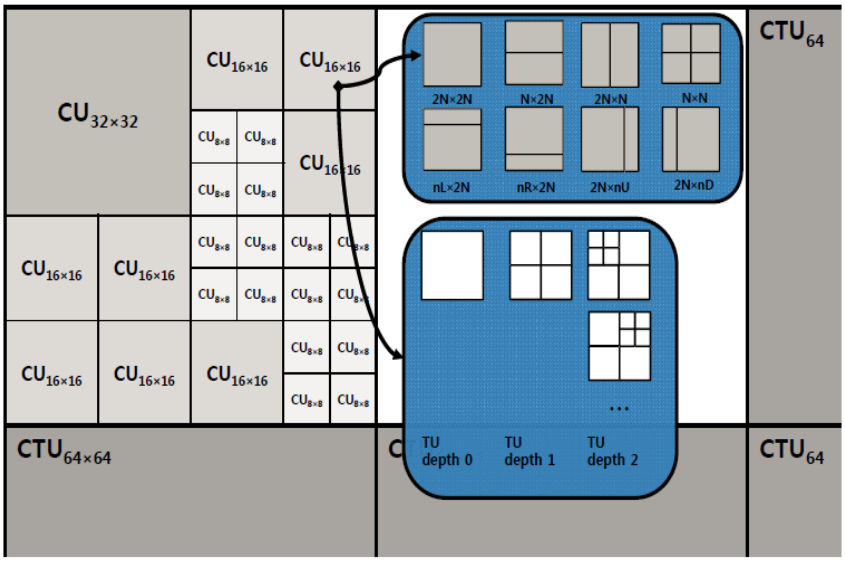

Figure 1-2 Quad-tree structure

1.2 Supported Services and Features

This user guide accompanies TI's implementation of HEVC Decoder on ARM.

This version of the codec has the following supported features of the standard:

- Shall support main profile up to level 5.
- Shall support main still picture profile decoding.
- Shall support I, P, and B frames decoding.
- Shall support all intra/inter prediction modes and block sizes.
- Shall support constrained intra prediction.
- Shall support scaling matrix.
- Shall support weighted prediction.
- Shall support PCM decoding.
- Shall support video resolutions up to 4096x2176 which are multiple of 2.
- Shall support progressive content decoding.
- Shall support multi-slice & multi-tile decoding.
- Shall support deblocking filter and SAO filtering.
- Shall support wavefront parallel processing.
- Shall support dependent slices.
- Shall support temporal layers.
- Shall support error robustness and error concealment.
- Shall support SEI and VUI parameter decoding.
- Shall support CRA (clean random access).
- Shall support long term reference frames.
- Shall support input and output low delay interface.
- Shall support decoder header only mode of operation.
- Shall support multi-core implementation.
- Shall support API at Frame level granularity.
- Shall support decoder data in YUV 4:2:0 planar format.
- Shall output decoded YUV compliant to ITU-T recommendation HEVC/H265

This page is intentionally left blank

Chapter 2

Installation Overview

This chapter provides a brief description on the system requirements and instructions for installing the codec component. It also provides information on building and running the sample test application.

Topic **Page**

2.1 System Requirements

This section describes the hardware and software requirements for the normal functioning of the codec component.

2.1.1 Hardware

ARM based EVM is used for testing.

2.1.2 Software

This project is compiled, assembled, archived, and linked using Linaro GCC ARM toolchain.

2.2 Installing the Component

The codec component is released as compressed archive. Following sub sections details on installation along with directory structure.

2.2.1 Installing the Component – Compressed archive

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a top-level directory called HEVC.D.ARM.01.00.00.xx (where “01.00.00.xx” is the version of the codec), under which directory named ARM_001 is created.

Figure 2-1 shows the sub-directories created in the ARM_001 directory in case of object release. *Figure 2-2* will show directory structure in case of Source release. Only “Src” is additional in source release compared to object only release package, remaining folders being same.

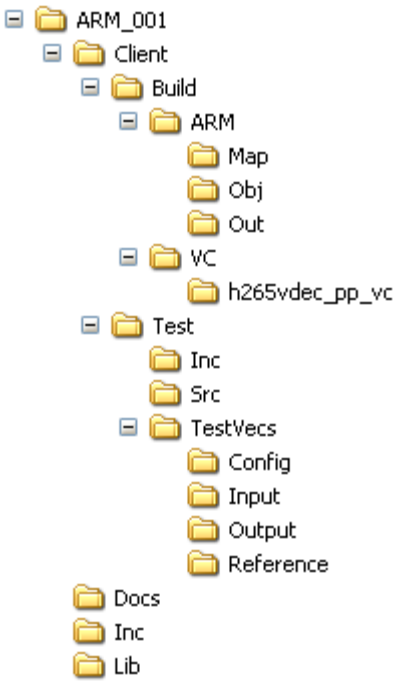

Figure 2-1 Component Directory Structure In case of Object Release

Table 2-1 provides a description of the sub-directories created in the ARM_001 directory.

Table 2-1 Component Directories in case of Object release

Sub-Directory	Description
\Client\Build\ARM\Map	Contains map file generated after building with Linaro ARM GCC toolchain
\Client\Build\ARM\Obj	Contains intermediate Object files generated after building test application with Linaro ARM GCC toolchain
\Client\Build\ARM\Out	Contains the final application executable file generated by the sample test application.
\Client\Build\VC\h265vdec_pp_vc	Contains project files to build stand alone test application for decoder on MSVC
\Client\Test\Inc	Contains standalone test application header files
\Client\Test\Src	Contains standalone test application source files
\Client \Test\TestVecs\Config	Contains sample configuration files for H265 decoder
\Client \Test\TestVecs\Input	Contains input test vectors
\Client \Test\TestVecs\Output	Contains output generated by the codec. It is empty directory as part of release

Sub-Directory	Description
\Client \Test\TestVecs\Reference	Contains read-only reference output to be used for cross-checking against codec output
\Docs	Contains user guide and release notes
\Inc	Contains header files, which allow interface to the codec library.
\Lib	Contains the library file named as h265vdec_pp_vc.lib, h265vdec_pp_lib.a for decoding the compressed video data

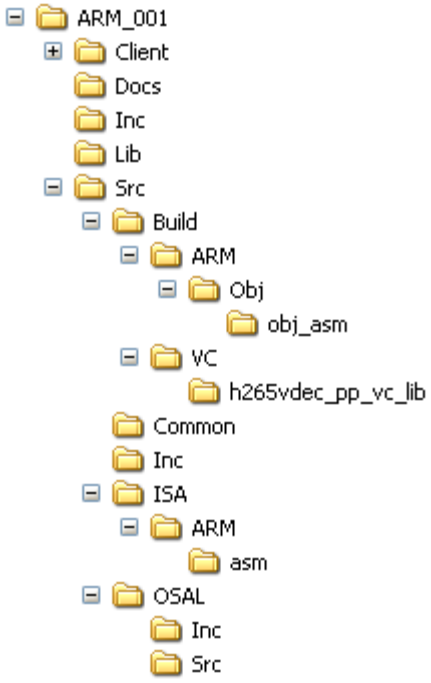

Figure 2-2 Component Directory Structure In case of Source Release

Table 2-2 below provides a description of the additional sub-directories, which are part of source release package compared to Object release directories (as in *Table 2-1*)

Table 2-2 Component Directories in case of Source release

Sub-Directory	Description
\Src\Build\ARM	Contains project files needed to build codec with Linaro ARM GCC toolchain
\Src\Build\ARM\Obj	Contains intermediate Object files generated for C files after building codec with Linaro ARM GCC toolchain
\Src\Build\ARM\Obj\obj_asm	Contains intermediate Object files generated for asm files after building codec with Linaro ARM GCC toolchain
\Src\Build\VC\h265vdec_pp_vc_lib	Contains project files needed to build codec with Microsoft Visual studio compiler
\Src\Common	Contains common source files needed to build codec
\Src\Inc	Contains common header files needed to build codec
\Src\ISA\ARM\asm	Contains hand written assembly files specific to ARM processor
\Src\OSAL\Inc	Contains operating system specific header files.
\Src\OSAL\Src	Contains operating system specific source files.

2.3 Before Building the Algorithm Library and Sample Test Application

This codec is accompanied by a sample test application. To build the sample test application, Linaro ARM GCC toolchain is required.

This version of the codec has been built with Linaro ARM GCC toolchain.

The version of the Linaro ARM GCC toolchain is 4.7-2013.03

2.3.1 Installing Linaro ARM GCC toolchain

Linaro ARM GCC toolchain version 4.7-2013.03 can be downloaded from the following website:

https://releases.linaro.org/13.03/components/toolchain/binaries

Download and extract the below zip file to /home/user/ on Linux machine
gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313_linux.tar.bz2

2.4 Building the Algorithm Library

Building algorithm library on Visual studio and Linux is specified in section 2.4.1 and 2.4.2 resepctively.

2.4.1 Building Algorithm Library on Visual studio

To build the algorithm library from source code in Visual Studio, follow these steps:

1) Verify that you have installed Microsoft Visual Studio 2008 Express Edition development environment. Open the source project “h265vdec_pp_vc_lib.vcproj” from “ARM_001\Src\Build\VC\h265vdec_pp_vc_lib\”.

2) This project contains two build configurations “Debug” and “Release”. “Debug” configuration will disable all the optimizations to debug the code. “Release” configuration will enable all the optimizations without exposing symbols. Please select “Debug” configuration.

3) Right click on the above project in Visual Studio IDE and select Build Project to build the algorithm library.

4) The built library, h265vdec_pp_vc.lib is available in the ARM_001\Lib sub-directory.

2.4.2 Building Algorithm Library on Linux

To build the algorithm library from source code on Linux, follow these steps:

1) Set the environment variable PATH to /bin path of ARM toolchain.

   Example:

   export PATH=$PATH:/home/user/gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313_linux/bin

2) Run the makefile to build the library.
ARM_001/Src/Build/ARM$ make

3) The built library, h265vdec_pp_lib.a is available in the ARM_001\Lib sub-directory.

2.5 Building Sample Test Application

2.5.1 Building Sample Test Application on Visual Studio

The sample test application that accompanies this codec component will run in Microsoft Visual Studio 2008 development environment. To build the sample test application in Visual studio 2008 Express edition, follow these steps:

1) Verify that you have installed Microsoft Visual Studio 2008 Express Edition development environment.

2) Verify that the following codec object libraries should exist in \Lib sub-directory.

3) h265vdec_pp_vc.lib: H265 Decoder.

4) Start the Visual studio 2008 Express Edition.

5) Select File->Open->Project/Solution and open “h265vdec_pp_vc.sln” located at “ARM_001\Client\Build\VC\h265vdec_pp_vc\”

6) Select Build->Build solution it builds the stand alone test application

2.5.2 Building the Sample Test Application on Linux

To build the sample test application on Linux, follow these steps:

1) Set the environment variable PATH to /bin path of ARM toolchain.

   Example:

   export PATH=$PATH:/home/user/gcc-linaro-arm-linux-gnueabihf-4.7-2013.03-20130313_linux/bin

2) Ensure the codec library h265vdec_pp_lib.a is available in the sub directory ARM_001/Lib

3) Run the makefile to generate the executable.
ARM_001/Client/Build/ARM$ make

4) The executable h265vdec_pp is generated in the ARM_001/Client/Build/ARM/Out sub-directory.

2.6 Configuration Files

This codec is shipped along with:

- Decoder configuration file (Testparams.cfg) – specifies the configuration parameters used by the test application to configure the Decoder.

2.6.1 Decoder Configuration File

The decoder configuration file, Testparams.cfg contains the configuration parameters required for the decoder. The Testparams.cfg file is available in the \Client\Test\TestVecs\Config sub-directory. A sample Testparams.cfg file is as shown.

```
# <ParameterName> = <ParameterValue> # Comment

###########################################################################
# Parameters
###########################################################################

################################### No. of threads ######################
NumThreads = 1
################################### Files ###############################
InputFile      = ..\..\..\Test\TestVecs\Input\airshow_352x288.265
OutputFile     = ..\..\..\Test\TestVecs\Output\airshow_352x288.yuv
ReferenceFile  = ..\..\..\Test\TestVecs\Reference\REF_airshow_352x288.yuv
ImageWidth          = 352   # Image width in Pels
ImageHeight         = 288   # Image height in Pels
FramesToDecode      = 10    # Number of frames to be coded
InputLowDelayMode   = 0     # 0->Entire frame, 1 ->Slice mode (Data sync
                              mode)
OutputLowDelayMode  = 0     # 0->Entire frame, 1 ->Number of CTU rows
                              (Data sync mode)
NumCTURows          = 0     # 0->Non-DataSync mode, Non-Zero positive when
                              OutputLowDelayMode is set to Data sync mode

MetadataType        = 0     # 0->No Metadata, 1-SEI, 2-VUI 3-SEI and VUI
#########################################################################
# Dynamic Parameters
#########################################################################
DecodeHeader         = 0    # 0-> Disable decode header mode, 1-> Enable
                              decode header mode
```

2.7 Running Sample Test Application

2.7.1 Running the Sample Test Application on Visual Studio

To run sample test application visual studio 2008 follow these steps:

1) Start the Visual studio 2008 Express Edition.
2) Select File->Open->Project/Solution and open “h265vdec_pp_vc.sln” located at “ARM_001\Client\Build\VC\h265vdec_pp_vc\”
3) Make sure code is built as specified in section 2.5.1.
4) Select Debug->Debug solution (F5) to run test application.
5) The sample test application takes the input files stored in the ARM_001\Client\Test\TestVecs\Input sub-directory, runs the codec, and uses the reference files stored in the ARM_001\Client\Test\TestVecs\Reference sub-directory to verify that the codec is functioning as expected.
6) On successful completion, the application displays the following messages for every display frame:
7) "Frame number <number> dumped"
8) The output is written to the specified file (this can then be manually compared against the reference).
9) On failure, the application prints the error message and exits.

2.7.2 Running the Sample Test Application on Linux

To run the sample test application on Linux, follow these steps:

1) Verify that executable is created in “ARM_001/Client/Build/ARM/Out” folder, after following steps in section 2.5.2.
2) Move the executable file h265vdec_pp from ARM_001/Client/Build/ARM/Out folder to the device.
3) Move the configuration file Testparams.cfg from ARM_001/Client/Test/TestVecs/Config folder to the device.
4) Move the input bit-stream file from ARM_001/Client/Test/TestVecs/Input folder to the device.
5) Move the reference file from ARM_001/Client/Test/TestVecs/Reference folder to the device.
6) Ensure configuration file Testparams.cfg and executable file h265vdec_pp are in the same path of the device.
7) After moving the files to the device set the path of the InputFile, OutputFile and ReferenceFile accordingly in the configuration file Testparams.cfg.

8) Run the sample test application that takes the input file and uses the reference file to verify that the codec is functioning as expected. If the reference file specified in the configuration file does not exist, it will not compare with the reference, but just save the decoded output.

   a) Command for running the sample test application:

      ./h265vdec_pp

9) On successful completion, the application displays the following messages for every display frame:

   " Frame number <number> dumped "

10) On failure, the application prints the error message and exits.

2.8 Uninstalling the Component

To uninstall the component, delete the codec directory from your hard disk.

Chapter 3

Sample Usage

This chapter provides a detailed description of the sample test application that accompanies this codec component.

3.1 Overview of the Test Application

The source files for this application are available in the \Client\Test\Src and \Client\Test\Inc sub-directories.

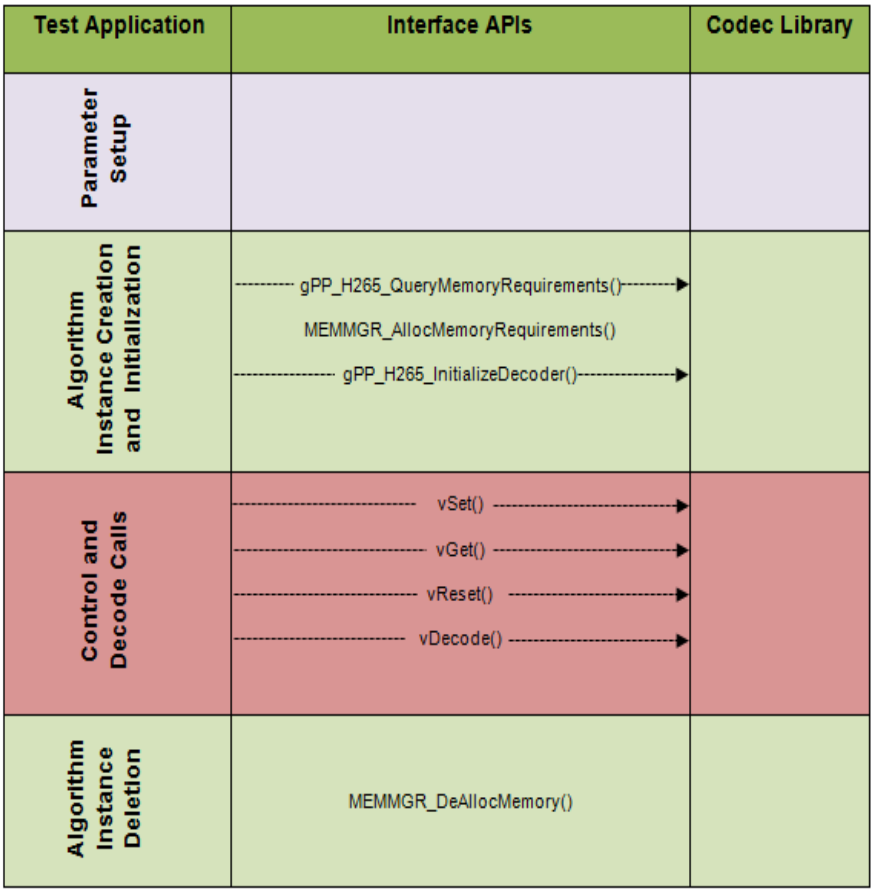

Figure 3-1 Test Application Sample Implementation

The test application is divided into four logical blocks:

- Parameter setup
- Algorithm instance creation and initialization
- Control and Decode call
- Algorithm instance deletion

3.1.1 Parameter Setup

Each codec component requires various codec configuration parameters to be set at initialization. For example, a video codec requires parameters such as video height, video width and so on. The test application obtains the required parameters from the Decoder configuration files.

In this logical block, the test application does the following:

1) Opens Decoder configuration file (Testparams.cfg), input file, and output/reference files.

2) Opens the Decoder configuration file, (Testparams.cfg) and reads the various configuration parameters required for the algorithm. For more details on the configuration files, see section 2.7.

3) Sets the `tPP_H265_CreateParams` structure based on the values it reads from the Testparams.cfg file.

4) Reads the input bit-stream into the application input buffer.

After successful completion of these steps, the test application does the algorithm instance creation and initialization.

3.1.2 Algorithm Instance Creation and Initialization

In this logical block, the test application accepts the various initialization parameters and returns an algorithm instance pointer. The following APIs implemented by the codec are called in sequence:

1) `gPP_H265_QueryMemoryRequirements()` - To query the algorithm about the number of memory records it requires and their sizes.

2) `MEMMGR_AllocMemoryRequirements()` - To allocate the memory as per the algorithm request. Memory is allocated by the test application.

3) `gPP_H265_InitializeDecoder()` - To initialize the algorithm with the memory structures provided by the application.

3.1.3 Control and Decode Call

After algorithm instance creation and initialization, the test application does the following:

1) Sets the dynamic parameters (if they change during run-time) by calling the `vSet()` function.

2) Sets the input and output buffer descriptors required for the `vDecode()` function call. The input and output buffer descriptors are obtained by calling the `vGet()` function.

3) Calls the `vDecode()` function to decode a single frame of data. The behavior of the algorithm can be controlled using various dynamic parameters (see Section 4.2.1.6).

Once the algorithm is initialized, there can be any ordering of control calls (`vGet(), vSet())` and decode call (`vDecode()`) functions. The following APIs are called in sequence:

1) `vGet() and vSet()` (optional) - To query the algorithm on status or setting of dynamic parameters.

2) `vDecode()` - To call the Decoder with appropriate input/output buffer and arguments information.

`vReset()` is called to reset the algorithm. All the fields in the internal data structures are reset and all internal buffers are flushed.

The do-while loop encapsulates frame level `vDecode()` call and updates the input buffer pointer every time before the next call. The do-while loop breaks off either when an error condition occurs or when the input buffer exhausts.

3.1.4 Algorithm Instance Deletion

Once decoding is complete, the test application must delete the current algorithm instance. The following API is is called:

1) `MEMMGR_DeAllocMemory()` - To free up the memory allocated for the algorithm.

3.2 Frame Buffer Management

3.2.1 Frame Buffer Input and Output

For each decode call, one frame buffer is provided with an unique buffer ID (`bufId`). After decode call this buffer is filled with the decoded data and it is either locked or given for display. When the codec gives the buffer(s) for display, its ID should be updated in free buffer ID array (`bufId[]`) so that application release the buffer(s). There is no distinction between DPB and display buffers. Application needs to ensure that it does not overwrite the buffers that are locked by the codec.

Note :

- Application can take the information retured by the function (`vGet()`) and change the size of the buffer passed in the next process call.
- For the first decode call, `nMaxHeight` and `nMaxWidth` are used for calculating buffer size. For subsequent decode calls (That is, after the first decode call, when the decoder gets to know the actual height and width from the headers) the actual height and width are used. Hence, this can be optionally used by the application to re allocate the buffer sizes, if required.

The frame buffer pointer given by the application and that returned by the algorithm may be different. `bufId` provides the unique ID to keep a record of the buffer given to the algorithm and released by the algorithm.

3.2.2 Frame Buffer Management by Application

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

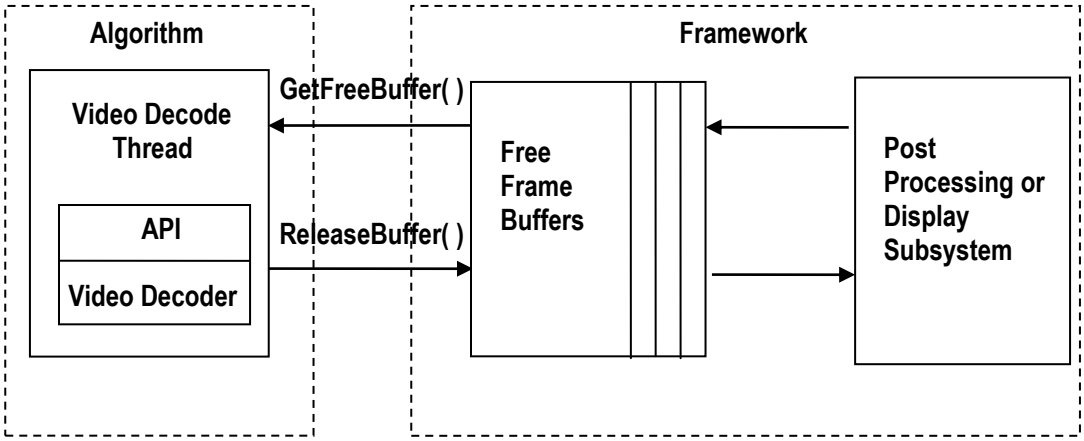

Figure 3-2 Interaction of Frame Buffers Between Application and Framework

The sample application also provides a prototype for managing frame buffers. It implements the following functions, which are defined in file h265vdec_pp_buffer_mngr.c provided along with test application.

- `BUFFMGR_Init() – 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()` - `BUFFMGR_ReInit` function allocates global luma and chroma buffers and allocates entire space to the first element. This element will be used in the first frame decode. After the picture height and width and its 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()` - `BUFFMGR_GetFreeBuffer` function searches for a free buffer in the global buffer array and returns the address of that element. Incase none of the elements are free, then it returns `NULL`.
- `BUFFMGR_ReleaseBuffer()` - `BUFFMGR_ReleaseBuffer` function takes an array of buffer-IDs which are released by the test application. 0 is not a valid buffer ID, hence this function moves until it encounters a buffer ID as zero or it hits the `MAX_BUFF_ELEMENTS`.

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

Chapter 4

API Reference

This chapter provides a detailed description of the data structures and interfaces functions used in the codec component.

Topic **Page**

4.1 Symbolic Constants and Enumerated Data Types

This section summarizes all the symbolic constants specified as either macros and/or enumerated C data types. For each symbolic constant, the semantics or interpretation of the same is also provided.

4.1.1 Common Data types

This section includes common enumerated data types:

Table 4-1 List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
tPP_DecodeMode	PP_H265_DECODE_ACCESSUNIT	Decode entire access unit including all the headers.
	PP_H265_PARSE_HEADER	Decode only header.
tPP_MetadataType	PP_H265_METADATAPLANE_NONE	Used to indicate no metadata is requested.
	PP_H265_METADATA_SEI_DATA	Used to indicate that SEI info metadata is requested.
	PP_H265_METADATA_VUI_DATA	Used to indicate that VUI info metadata is requested.
tPP_InputLowDelayMode	PP_H265_ENTIREINPUTFRAME	Used to indicate input low delay mode is in entire frame level mode.
	PP_H265_SLICEMODE	Used to indicate input low delay mode is in slice level mode.
tPP_OutputLowDelayMode	PP_H265_ENTIREOUTPUTFRAME	Used to indicate output low delay mode is in entire frame level mode.
	PP_H265_NUMCTUROWS	Used to indicate output low delay mode is in CTU rows level mode.

Table 4-2 Decoder Error Codes

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
PP_H265VDEC_Errors	PP_H265_ERR_UNSUPPORTED	This error is set when the decoder does not support a specific parameter.
	PP_H265_ERR_NO_NALU_BEGIN_FOUND	This error is set when start code used to identify the beginning of a NAL unit is not found.
	PP_H265_ERR_NO_NALU_END_FOUND	This error is set when start code used to identify the end of a NAL unit is not found.
	PP_H265_ERR_INVALID_NAL_UNIT_TYPE	This error is set when the NAL unit type is not a valid type.
	PP_H265_ERR_INSUFFICIENT_BUFFER	This error is set when the input data provided is not sufficient to produce one frame of data.
	PP_H265_ERR_DATA_SYNC	This error is set when the output data sync mode is enabled and the picture is having different display and decoding order.
	PP_H265_ERR_CRITICAL	This error is set when the bit-stream has an error and further decoding is not possible.
	PP_H265_ERR_NO_VPS	This error is set when the bit-stream has no VPS header.
	PP_H265_ERR_VPS	This error is set when the bit-stream has an error in VPS header.
	PP_H265_ERR_NO_SPS	This error is set when the bit-stream has no SPS header.
	PP_H265_ERR_SPS	This error is set when the bit-stream has an error in SPS header.
	PP_H265_ERR_NO_PPS	This error is set when the bit-stream has no PPS header.
	PP_H265_ERR_PPS	This error is set when the bit-stream has an error in PPS header.
	PP_H265_ERR_SLICELOSS	This error is set when a slice is missing in the bit-stream.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	PP_H265_ERR_SLICEHDR	This error is set when the bit-stream has an error in Slice header.
	PP_H265_ERR_SLICEDATA	This error is set when the bit-stream has an error in Slice data.
	PP_H265_ERR_RANDOM_ACCESS_SKIP	This error is set when the bit-stream has an error in poc id around random access point.
	PP_H265_ERR_REFPIC_NOT_FOUND	This error is set when the reference picture is not found from the decoder picture buffer.
	PP_H265_ERR_META_DATA	This error is set when the bit-stream has an error in SEI or VUI header.

4.1.2 Common Multi-Core Data types

This section describes common data types used for Multithread operations. Following data types are described in current section.

- eH265_multiThreadTask

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
eH265_multiThreadTask	kH265_TASK_MASTER	0	Master thread task ID.
	kH265_TASK_SLAVE	1	Slave thread task ID.

4.2 Data Structures

This section describes the defined data structures that are common across codec classes. These data structures can be extended to define any implementation specific parameters for a codec component.

4.2.1 Common Data Structures

This section includes the following common data structures:

- `tPPInFrame_Buff`
- `tPPOutFrame_Buff`
- `tPPInput_BitStream`
- `tPPBaseDecoder`
- `tPP_H265_CreateParams`
- `tPP_H265_DynamicParams`
- `tPPDecParam_Status`
- `tPPYUVPlanarDisplayFrame`
- `tPPBaseVideoFrame`

4.2.1.1 tPPInFrame_Buff

‖ Description

This structure defines the input frame buffer details.

‖ Fields

Field	Data Type	Input/ Output	Description
*buf[]	tPPu8	Input	Array of input buffer pointers
bufId	tPPi32	Input	Application passes this ID to algorithm and decoder will attach this ID to the corresponding output frames. This is useful in case of re-ordering (for example, B frames). If there is no re-ordering, bufId field in the tPPOutFrame_Buff data structure will be same as this ID.

4.2.1.2 tPPOutFrame_Buff

‖ Description

This structure defines the output frame buffer details.

‖ Fields

Field	Data Type	Input/ Output	Description
bufId[PP_MAX_REF_FRAME_COUNT]	tPPi32	Output	This is an array of buffer IDs corresponding to the frames that have been unlocked in the current decode call.

4.2.1.3 tPPInput_BitStream

‖ Description

This structure defines the input bitstream buffer details.

‖ Fields

Field	Data Type	Input/ Output	Description
*nBitStream	tPPu8	Input	Pointer to input bitstream buffer.
nBufLength	tPPi32	Input	Length of the input bitstream buffer in bytes.

Note:

For HEVC Decoder, the buffer details are:

- Number of input buffer required is 1.
- Number of output buffers required is 3 (one for Y plane, 1 for Cb and 1 for Cr) , if no metadata is requested by the application.
- If metadata is requested by the application, then see the Appendix A for buffer details.
- For frame mode of operation, there is no restriction on input buffer size except that it should contain atleast one frame of encoded data.

4.2.1.4 tPPBaseDecoder

‖ Description

This structure contains pointers to all the decoder interface functions.

‖ Fields

Field	Data Type	Input/ Output	Description
`vDecode`	`(*)()`	Input	Pointer to the decode function. See section 4.4 for more information.
`vReset`	`(*)()`	Input	Pointer to the reset function. See section 4.4 for more information
`vSet`	`(*)()`	Input	Pointer to the set function which sets parameters. See section 4.4 for more information
`vGet`	`(*)()`	Input	Pointer to the get function which gets the parameters information. See section 4.4 for more information

4.2.1.5 tPP_H265_CreateParams

‖ Description

This structure defines the creation parameters for an algorithm instance object. For the default and supported values, see *Table 4-3*.

‖ Fields

Field	Data Type	Input/ Output	Description
nMaxWidth	tPPi32	Input	Maximum video width to be supported in pixels.
nMaxHeight	tPPi32	Input	Maximum video height to be supported in pixels.
nNumThreads	tPPi32	Input	Total number of threads.
nInputLowDelayMode	tPPi32	Input	Input low delay interface mode. When it is set, decoder takes input bitstream data at slice level/fixed length through call back function. In this version of codec, fixed length mode is not supported.
nOutputLowDelayMode	tPPi32	Input	Output low delay interface mode. When it is set, decoder gives output data at row level through call back function.
nNumCTURows	tPPi32	Input	Number of output CTU rows. This parameter is valid only when nOutputLowDelayMode is set to PP_H265_NUMCTUROWS
nMetadataType	tPPi32	Input	Type of metadata. See eH265_MetadataType enumeration for details.

4.2.1.6 tPP_H265_DynamicParams

‖ Description

This structure defines the run-time parameters for an algorithm instance object. For the default and supported values, see *Table 4-4*

‖ Fields

Field	Data Type	Input/ Output	Description
nDecodeHeader	tPPi32	Input	If the field is set to: ❑ 0 (PP_H265_DECODE_ACCESSUNIT) - Decode entire frame including all the headers ❑ 1 (PP_H265_PARSE_HEADER) - Decode only header
fOutputLowDelayFxn	gH265_DataSyncPutFxn	Input	Function pointer to produce output frame data at sub frame level (Data sync call back function pointer for OuputLowDelayFunction).
fInputLowDelayFxn	gH265_DataSyncGetFxn	Input	Function pointer to receive partial bitstream data at sub frame level (Data sync call back function pointer for InputLowDelayFunction).

4.2.1.7 tPPDecParam_Status

‖ Description

This structure defines parameters that describe the status of an algorithm instance object.

‖ Fields

Field	Data Type	Input/ Output	Description
nTotalFrameSize	tPPi32	Output	Frame size in pixels.
nPicWidth	tPPi32	Output	Picture width in pixels.
nPicHeight	tPPi32	Output	Picture height in pixels.
nProfile	tPPi32	Output	Profile id.
nInputLowDelayMode	tPPi32	Output	Input low delay interface mode. When it is set, decoder takes input bitstream data at slice level/fixed length through call back function.
nOutputLowDelayMode	tPPi32	Output	Output low delay interface mode. When it is set, decoder gives output data at row level through call back function.

Field	Data Type	Input/ Output	Description
nNumCTURows	tPPi32	Output	Number of output CTU rows.
nMetadataType	tPPi32	Output	Type of metadata. See eH265_MetadataType enumeration for details.
nDecodeHeader	tPPi32	Output	If the field is set to: ❑ 0 (PP_H265_DECODE_ACCESSUNIT) - Decode entire frame including all the headers. ❑ 1 (PP_H265_PARSE_HEADER) - Decode only header.
fOutputLowDelayFxn	gH265_DataSyncPutFxn	Output	Function pointer to produce output frame data at sub frame level (Data sync call back function pointer for OuputLowDelayFunction).
fInputLowDelayFxn	gH265_DataSyncGetFxn	Output	Function pointer to receive partial bitstream data at sub frame level (Data sync call back function pointer for InputLowDelayFunction).
nError	tPPi32	Output	Indicates the error type, if any.

4.2.1.8 tPP_H265_DataSyncDesc

‖ Description

This structure provides the descriptor for the chunk of data being transferred in one call to InputLowDelayFunction or OutputLowDelayFunction.

‖ Fields

Field	Data Type	Input/ Output	Description
nNumBlocks	tPPi32	Output	Number of blocks given out by decoder in output data sync call back function. Each block is having 16 lines of video. nNumBlocks = (k*tpp_H265_CreateParams::nNumCTURows) where k = 1, 2, or 3 depending on CTU size (16x16, 32x32, or 64x64) repectively. Example: if nNumCTURows = 2 nNumBlocks = 2 (CTU size: 16x16) nNumBlocks = 4 (CTU size: 32x32) nNumBlocks = 8 (CTU size: 64x64)
nSize	tPPi32	Input	Size of the input bitstream data provided to decoder in input data sync call back function.

4.2.1.9 tPPYUVPlanarDisplayFrame

‖ Description

This structure gives the display frame information.

‖ Fields

Field	Data Type	Input/ Output	Description
nBaseFrame	tPPBaseVideo Frame	Output	Base video frame structure with frame size and padding information.
*pLum	tPPu8	Output	Pointer to luma display buffer
*pCb	tPPu8	Output	Pointer to chroma display buffer (Cb)
*pCr	tPPu8	Output	Pointer to chroma display buffer (Cr).

4.2.1.10 tPPBaseVideoFrame

‖ Description

This structure contains frame size and padding information.

‖ Fields

Field	Data Type	Input/ Output	Description
nWidth	tPPi32	Output	Actual width of the video frame.
nHeight	tPPi32	Output	Actual height of the video frame.
nIsPadded	tPPi32	Output	Flag to indicate whether frame is padded or not.
nExWidth	tPPi32	Output	This field is used to indicate the padded width. It is used as frame pitch to store the frame.

4.2.2 Common Multi-thread Data Structures

This section descibes below data structures which are used for multicore operations. These data structures are common across all codecs.

- tPPMultiThreadParams

4.2.2.1 tPPMultiThreadParams

Description

This structure contains control parameters that are used for multi-thread program flow. All the fields must be set by the application before calling codec instance creation.

Fields

Field	Data Type	Input/ Output	Description
nThreadID	tPPi32	Input	Thread identification variable.
nTaskID	eH265_multiThreadTask	Input	Thread task identification variable. See `eH265_multiThreadTask` enumeration for more details.
nNumThreads	tPPi32	Input	Total number of threads.

4.3 Default and Supported Values of Parameters

This section provides the default and supported values for the following data structures:

- tPP_H265_CreateParams
- tPP_H265_DynamicParams

Table 4-3 Default and Supported Values for tPP_H265_CreateParams

Field	Default Value	Supported Value
nMaxWidth	1920	64 < = nMaxWidth < = 4096
nMaxHeight	1088	64 < = nMaxHeight < = 2176
nNumThreads	1	1 to 4
nInputLowDelayMode	PP_H265_ENTIREINPUTFRAME	❑ PP_H265_ENTIREINPUTFRAME ❑ PP_H265_SLICEMODE In this version of codec, fixed length mode is not supproted
nOutputLowDelayMode	PP_H265_ENTIREOUTPUTFRAME	❑ PP_H265_ENTIREOUTPUTFRAME ❑ PP_H265_NUMCTUROWS
nNumCTURows	Don't Care	Any positive number when nOutputLowDelayMode is set to PP_H265_NUMCTUROWS
nMetadataType	PP_H265_METADATAPLANE_NONE	❑ PP_H265_METADATAPLANE_NONE ❑ PP_H265_METADATA_SEI_DATA ❑ PP_H265_METADATA_VUI_DATA

Note:

- During codec creation, maxHeight and maxWidth as specified in above table are allowed. Note that maxHeight and maxWidth should be always greater than or equal to image width and image height.

Table 4-4 Default and Supported Values for tPP_H265_DynamicParams

Field	Default Value	Supported Value
nDecodeHeader	PP_H265_DECODE_ACCESSUNIT	❑ PP_H265_DECODE_ACCESSUNIT ❑ PP_PARSE_HEADER
fOutputLowDelayFxn	NULL	Valid (Non-NULL) function pointer
fInputLowDelayFxn	NULL	Valid (Non-NULL) function pointer

4.4 Interface Functions

This section describes the Application Programming Interfaces (APIs) used in the H.265 Decoder. The APIs are logically grouped into the following categories:

- **Creation** – `gPP_H265_QueryMemoryRequirements()`, `MEMMGR_AllocMemoryRequirements()`
- **Initialization** – `gPP_H265_InitializeDecoder()`
- **Control** – `vGet(), vSet(), vReset()`
- **Data processing** – `vDecode()`
- **Termination** – `MEMMGR_DeAllocMemory()`

You must call these APIs in the following sequence:

1) `gPP_H265_QueryMemoryRequirements()`
2) `MEMMGR_AllocMemoryRequirements()`
3) `gPP_H265_InitializeDecoder()`
4) `vDecode()`
5) `MEMMGR_DeAllocMemory()`

`vGet()` and `vSet()` can be called any time after calling the `gPP_H265_InitializeDecoder()` API.

`vReset()` is called to reset the algorithm. All the fields in the internal data structures are reset and all internal buffers are flushed.

4.4.1 Creation APIs

Creation APIs are used to create an instance of the component. The term creation could mean allocating system resources, typically memory.

‖ Name

`gPP_H265_QueryMemoryRequirements()` – determine the attributes of all buffers that an algorithm requires

‖ Synopsis

```
tPPResult gPP_H265_QueryMemoryRequirements(
                    tPPQueryMemRecords *apQueryMemRecords,
                    tPP_H265_CreateParams *apCreateParams,
                    tPPu32 *apNumMemEntries);
```

‖ Arguments

```
tPPQueryMemRecords *apQueryMemRecords; /* output array of
memory records */

tPP_H265_CreateParams *apCreateParams; /* algorithm
specific attributes */

tPPu32 *apNumMemEntries; /* Number of buffers (memtab
entries) required */
```

‖ Return Value

```
SC_PP_SUCCESS; /* status indicating success */

EC_PP_FAILURE; /* status indicating failure */
```

‖ Description

`gPP_H265_QueryMemoryRequirements()` returns a table of memory records that describe the allotedHandle address and size of all buffers required by the algorithm. After calling this function, size field of the memory records structure is filled up by the algorithm. If successful, this function returns `SC_PP_SUCCESS` indicating the size of all the required buffers are updated properly by the algorithim otherwise it returns `EC_PP_FAILURE.`

The first argument to `gPP_H265_QueryMemoryRequirements()` is a pointer to a memory space of size `PP_H265DEC_MAX_MEMTAB * sizeof (tPPQueryMemRecords)` where `PP_H265DEC_MAX_MEMTAB` is the maximum number of buffers and `tPPQueryMemRecords` is the buffer-descriptor structure.

The second argument to `gPP_H265_QueryMemoryRequirements()` is a pointer to a structure that defines the creation parameters.

The third argument to `gPP_H265_QueryMemoryRequirements()` is a variable which specifies the number of buffers required by the algorithm.

‖ See Also

`MEMMGR_AllocMemoryRequirements()`

‖ Name

`MEMMGR_AllocMemoryRequirements()` – allocate memory as per the algorithm request

‖ Synopsis

```
tPPResult MEMMGR_AllocMemoryRequirements(
                    tPPQueryMemRecords *apQueryMemRecords,
                    tPPu32 aNumMemTabEntries);
```

‖ Arguments

```
tPPQueryMemRecords *apQueryMemRecords; /* array of memory
records */

tPPu32 aNumMemTabEntries; /* Number of buffers (memtab
entries) requested by the algorithm */
```

‖ Return Value

```
SC_PP_SUCCESS; /* status indicating success */

EC_PP_OUT_OF_MEMORY; /* status indicating insufficient
memory */
```

‖ Description

`MEMMGR_AllocMemoryRequirements()` allocates memory as per the algorithm request. If all the buffers are allocated as per the request, this function returns `SC_PP_SUCCESS` otherwise it returns `EC_PP_OUT_OF_MEMORY` indicating buffer allocation failure.

The first argument to `MEMMGR_AllocMemoryRequirements()` is a pointer to a memory space of size `PP_H265DEC_MAX_MEMTAB * sizeof (tPPQueryMemRecords)` where `PP_H265DEC_MAX_MEMTAB` is the maximum number of buffers and `tPPQueryMemRecords` is the buffer-descriptor structure.

The second argument to `gPP_H265_QueryMemoryRequirements()` is a variable which specifies the number of buffers requested by the algorithm.

After calling this function, all the buffers are allocated and the corresponding buffer pointers are updated to memory records buffer descriptor structure.

‖ See Also

`gPP_H265_QueryMemoryRequirements(), MEMMGR_DeAllocMemroy()`

4.4.2 Initialization API

Initialization API is used to initialize an instance of the algorithm. The initialization parameters are defined in the `tPP_H265_CreateParams` structure (see section 4.2 for details).

‖ Name

`gPP_H265_InitializeDecoder()` – initialize an algorithm instance

‖ Synopsis

```
tPPResult gPP_H265_InitializeDecoder(
                    tPPBaseDecoder **apBase,
                    tPPQueryMemRecords *apQueryMemRecords,
                    tPP_H265_CreateParams *apCreateParams);
```

‖ Arguments

```
tPPBaseDecoder **apBase; /* pointer to algorithm instance
handle*/

tPPQueryMemRecords *apQueryMemRecords; /* pointer to
memory records buffer*/

tPP_H265_CreateParams *apCreateParams; /*algorithm init
parameters */
```

‖ Return Value

```
SC_PP_SUCCESS; /* status indicating success */

EC_PP_FAILURE; /* status indicating failure */
```

‖ Description

`gPP_H265_InitializeDecoder()` performs all initializations necessary to complete the run time creation of an algorithm instance object. After a successful return from `gPP_H265_InitializeDecoder()`, the instance object is ready to be used to process data. This function returns `SC_PP_SUCCESS` indicating the algorithm initialization is done successfully otherwise it returns `EC_PP_FAILURE`.

The first argument to `gPP_H265_InitializeDecoder()` is a pointer to algorithm instance handle. This value is initialized to the allotedHandle field of `apQueryMemRecords[0]`.

The second argument is a pointer to table of memory records that describe the allotedHandle address and size of all buffers allocated for an algorithm instance. The number of initialized records is identical to the number updated by the API `gPP_H265_QueryMemoryRequirements()`.

The last argument is a pointer to a structure that defines the algorithm initialization parameters.

4.4.3 Control API

Control API is used for controlling the functioning of the algorithm instance during run-time. This is done by changing the status of the controllable parameters of the algorithm during run-time.

‖ Name

`vSet()` – change run time parameters and query the status

‖ Synopsis

```
tPPResult (*vSet) (tPPBaseDecoder *apBase, tPPu32 anFlag,
tPPi32 anVal, tPP_H265_DynamicParams *apDynamicParams);
```

‖ Arguments

```
tPPBaseDecoder *apBase; /* algorithm instance handle */

tPPu32 anFlag; /* specifies the parameter to be set */

tPPi32 anVal; /* specifies the value of the parameter */

tPP_H265_DynamicParams *apDynamicParams; /* pointer to dynamic
parameters structure */
```

‖ Return Value

```
SC_PP_SUCCESS; /* status indicating success */

EC_PP_FAILURE; /* status indicating failure */
```

‖ Description

This function sets the run time parameters of an algorithm instance. `vSet()` must only be called after a successful call to `gPP_H265_InitializeDecoder()`. When the parameter is set successfully, this function returns `SC_PP_SUCCESS` otherwise it returns `EC_PP_FAILURE`.

The first argument to `vSet()` is a handle to an algorithm instance.

The second argument specifies the control command for setting the corresponding parameter.

When it is set to `PP_SET_DPB_FLUSH`, dpb flush flag is set.

When it is set to `PP_SET_DECODE_HDR`, decoder header parameter is set.

When it is set to `PP_SET_RES_INV`, width and height parameters are reset.

When it is set to `PP_SET_DATASYNC`, input or output or both low delay interface call back function pointers are set.

The third argument is the value used for setting the parameter when `anFlag` is set to `PP_SET_DPB_FLUSH` and `PP_SET_DECODE_HDR`. In other cases it is ignored.

The last argument is a pointer to dynamic parameters structure.

‖ See Also

`vGet()`

‖ Name

`vGet()` – query the status of the run time parameters

‖ Synopsis

```
tPPResult (*vGet) (tPPBaseDecoder *apBase, tPPu32 anFlag,
void *apVal);
```

‖ Arguments

```
tPPBaseDecoder *apBase; /* algorithm instance handle */

tPPu32 anFlag; /* specifies the parameter to be get */

void *apVal; /* pointer to the parameter value or

                pointer to the tPPDecParam_Status */
```

‖ Return Value

```
SC_PP_SUCCESS; /* status indicating success */

EC_PP_FAILURE; /* status indicating failure */
```

‖ Description

This function queries the status of the algorithm's parameters. `vGet()` must only be called after a successful call to `gPP_H265_InitializeDecoder()`. When the parameter is get successfully, this function returns `SC_PP_SUCCESS` otherwise it returns `EC_PP_FAILURE`.

The first argument to `vGet()` is a handle to an algorithm instance.

The second argument is the parameter to be queried.

When it is set to `PP_GET_PARAMSTATUS`, all the `tPP_H265_CreateParams` and `tPP_H265_DynamicParams` are updated to `tPPDecParam_Status` structure.

When it is set to `PP_GET_BUFSTATUS`, buffer status is updated.

When it is set to `PP_GET_ERRORSTATUS`, error status is updated.

The third argument is the variable to which status of the parameter is updated.

‖ See Also

`vSet()`

‖ Name

`vReset()` – reset the algorithm

‖ Synopsis

```
tPPResult (*vReset) (tPPBaseDecoder *apBase);
```

‖ Arguments

```
tPPBaseDecoder *apBase; /* algorithm instance handle */
```

‖ Return Value

```
SC_PP_SUCCESS; /* status indicating success */

EC_PP_FAILURE; /* status indicating failure */
```

‖ Description

This function resets the algorithm. All the field in the internal data structures are reset and all internal buffers are flushed. When the algorithm is reset successfully, this function returns `SC_PP_SUCCESS` otherwise it returns `EC_PP_FAILURE`.

The argument to `vReset()` is a handle to an algorithm instance.

4.4.4 Data Processing API

Data processing API is used for processing the input data.

‖ Name

`vDecode()` – basic decoding call

‖ Synopsis

```
tPPResult (*vDecode)(tPPBaseDecoder *apBase,
tPPInput_BitStream *apInBitStream, tPPInFrame_Buff
*apH265_dec_InBuff, tPPOutFrame_Buff *apH265_dec_OutBuff,
tPPYUVPlanarDisplayFrame *apFrame, tPPMultiThreadParams
*nMTParam);
```

‖ Arguments

```
tPPBaseDecoder *apBase; /* algorithm instance handle */

tPPInput_BitStream *apInBitStream; /* input bitstream
structure */

tPPInFrame_Buff *apH265_dec_InBuff; /* input buffer
structure */

tPPOutFrame_Buff *apH265_dec_OutBuff; /* output buffer ID
structure */

tPPYUVPlanarDisplayFrame *apFrame; /* output buffer
structure for display */

tPPMultiThreadParams *nMTParam; /* multithread param
structure */
```

‖ Return Value

```
SC_PP_SUCCESS; /* status indicating success */

EC_PP_FAILURE; /* status indicating failure */
```

‖ Description

This function does the basic decoding.

The first argument to `vDecode()` is a handle to the algorithm instance.

The second argument is a pointer to the input bitstream structure. This structure has information about bitstream buffer pointer and the buffer length (see `tPPInput_BitStream` data structure for details).

The third argument is a pointer to the input buffer data structure. (see `tPPInFrame_Buff` data structures for details). This structure has the pointer to the buffer allocated by the buffer manager to the decoder. The decoder stores the current decoded frame in this location.

The fourth argument is a pointer to the output buffer data structure (see `tPPOutFrame_Buff` data structures for details). This structure has the pointer to the buffer to be freed by the decoder. The application sets ‘0’ in these locations as BuffIDs. The decoder sets the BuffID of the buffer which needs to be freed.

The fifth argument is a pointer to the display buffer data structure (see `tPPYUVPlanarDisplayFrame` data structures for details). This structure has the display buffer pointers for the Y, U, and V components, actual width and height of the video frame, padding information, and width of the padded buffer.

The last argument is a pointer to the multithread parameters structure (see `tPPMultiThreadParams` data structures for details). This structure gives the information about number of threads, thread ID and thread task ID (master or slave).

4.4.5 Termination API

Termination API is used to terminate the algorithm instance and free up the memory space that it uses.

‖ Name

`MEMMGR_DeAllocMemory()` – free up all the buffers allocated for the algorithm.

‖ Synopsis

```
tPPResult MEMMGR_DeAllocMemory(
                    tPPQueryMemRecords *apQueryMemRecords);
```

‖ Arguments

```
tPPQueryMemRecords *apQueryMemRecords; /* pointer to a
buffer having an array of memory records */
```

‖ Return Value

```
SC_PP_SUCCESS; /* status indicating success */

EC_PP_FAILURE; /* status indicating failure */
```

‖ Description

All the buffers allocated by the `MEMMGR_AllocMemoryRequirements()` are freed by the `MEMMGR_DeAllocMemory()` function. If successful, it returns `SC_PP_SUCCESS` indicating all the buffers are freed otherwise it returns `EC_PP_FAILURE`.

The argument is a pointer to table of memory records that describe the allotedHandle address and size of all buffers allocated for algorithm instance.

This page is intentionally left blank

Chapter 5

Frequently Asked Questions

This chapter provides answers to few frequently asked questions related to using this decoder.

5.1 Code Build and Execution

Question	Answer
Application returns an error saying “Unable to open Input File” while running the host test application	Ensure that input file (*.265) path provided is correct in decoder configuration file. If the application is accessing input file from network, ensure that the network connectivity is stable.

5.2 Tools Version

Question	Answer
What are the tools and versions used to run the standalone codec?	To run the codec on standalone setup, below tools are used Linaro ARM GCC toolchain is 4.7-2013.03

5.3 Algorithm Related

Question	Answer
Does codec support XDM/XDAIS interface?	Codec does not support XDM/XDAIS interface
What are the profiles supported in this version of decoder?	This version of decoder supports Main Profile and Main Still Profile.
What is the maximum level supported by this decoder?	The decoder supports the level up to 5.
What are the output frame formats supported?	This version supports only YUV420 planar output buffer format.
What are the resolutions supported?	This version supports video resolutions up to 4096x2176 which are multiple of 2.

Question	Answer
Does this version of decoder support decoding multiple slices in a frame?	Yes.
Does this version of decoder support decoding multiple tiles in a frame?	Yes.
Does this version of decoder support wavefront parapllel processing?	Yes.
Does this version of decoder support B frames decoding?	Yes.
Does this version of decoder support SEI and VUI parmeters decoding?	Yes.
Can the decoder be run on any OS?	No. It runs on armv7 architecture based ARM platforms with Linux kernel.
Does this version of decoder support 10-bitdepth?	No.
Does this version of decoder support low delay interface (Data sync)?	Yes. It supports low delay interface (data sync) at both input and output. In input low delay interface slice mode is supported and fixed length mode is not supported.
What is granularity of the decode call?	The decoder supports only frame level decoding API.
The decode call returns error, what are the possible reasons?	The following are few of reasons for the error: The input or output pointers are null The input or output buffer sizes are not sufficient or incorrect Run time error occurred during decoding of the frame

Appendix A

Meta Data Support

This version of the decoder supports writing out the parsed SEI and VUI data into application provided buffers. If SEI and VUI is present in the stream for this frame, the parsed data is given back to the application.

This feature can be enabled/disabled through create time parameters `tPP_H265_CreateParams::nMetadataType`.

`nMetadataType` can take following values.

Enumeration	Value
`PP_H265_METADATAPLANE_NONE`	0
`PP_H265_METADATA_SEI_DATA`	1
`PP_H265_METADATA_VUI_DATA`	2

If user wants to get the SEI data, then
`tPP_H265_CreateParams::nMetadataType` should be set to `PP_H265_METADATA_SEI_DATA`.
If user wants to get the VUI data, then
`tPP_H265_CreateParams::nMetadataType` should be set to `PP_H265_METADATA_VUI_DATA`.
If both SEI and VUI data are needed, then
`tPP_H265_CreateParams::nMetadataType` should be set to `(PP_H265_METADATA_SEI_DATA | PP_H265_METADATA_VUI_DATA)`

If user does not want to use any meta data then
`tPP_H265_CreateParams::nMetadataType` should be set to `PP_H265_METADATAPLANE_NONE`.

Application provides the buffers required for metadata.

- If both SEI and VUI data is requested during create, then the number of input buffers needed is 5 (3 for Y, U, and V data, one each for SEI and VUI).

- If only one of SEI or VUI data is requested, then the number of input buffers needed is 4.

The buffer pointers for the metadata need to be supplied as below during decode call:

- When the application calls the `vDecode()` function, the buffer pointers where SEI and VUI data should be stored needs to be provided to the codec through `[tPP_InFrame_Buff H265_dec_InBuff]` structure members.
  - `H265_dec_InBuff.buf[3]` -> Buffer allocated for SEI
  - `H265_dec_InBuff.buf[4]` -> Buffer allocated for VUI

- Codec assumes the same fixed order as mentioned above and updates the SEI/VUI metadata.

Decoder parses metadata in the current decode call and returns in the same decode call. This means, effectively meta data will be given out in decode order [Not in Display Order]. If application is interested in display order, it should have logic to track based on input and output ID.

Appendix B

Low Delay Interface

B.1 Brief Description

Low delay interface (Sub frame level data synchronization) between decoder and application is implemented in this release at both input and output level.

At decoder input level (Bit Stream), slice mode of operation is supported where in individual NALs can be given.

At decoder output level, decoder can give out reconstructed rows of CTU, instead of waiting until the entire frame is reconstructed.

B.2 Details of using Low Delay Interface at output side

This section explains the low delay interface details at the output side.

This feature can be enabled/disabled through create time parameters `tPP_H265_CreateParams:: nOutputLowDelayMode`.

Creation time parameter related to low delay interface (sub frame level data communication) for output data of video decoder:

Parameter Name	**Details**	**Valid values**	
tpp_H265_CreateParams::nOutputLowDelayMode	Defines the mode of producing the output frame data.	*PP_H265_ENTIREOUTPUTFRAME*	Entire frame data is produced by decoder for display
		PP_H265_NUMCTUROWS	Frame data is given in unit of number of CTU rows. Number of CTU rows is converted to number of 16 video lines based on CTU size and the values is updated to the parameter tpp_H265_DataSyncDesc::nNumBlocks
tpp_H265_CreateParams::nNumCTURows	Unit of output data	Don't care if tpp_H265_CreateParams::nOutputLowDelayMode == PP_H265_ENTIREOUTPUTFRAME If	

		tpp_H265_CreateParams::nOutputLowDelayMode == PP_H265_NUMCTUROWS then it defines the frequency at which decoder should inform to application about data availability. For example nNumCTURows = 2 means that after every 2 CTU rows (2*k*16 lines) availability in display buffer, decoder should inform to application. The value of k can be 1,2, or 4 depending on CTU size (16x16, 32x32, or 64x64) repectively.

Dynamic parameters related to low delay interface (sub frame level data communication) for output data of video decoder:

Parameter Name	**Details**	**Valid values**
tpp_H265_DynamicParams::fOutputLowDelayFxn	This function pointer is provided by the app/framework to the video decoder. The decoder calls this function when sub-frame data has been put into an output buffer and is available.	*Any non-NULL value if nOutputLowDelayMode != PP_H265_ENTIREOUTPUTFRAME*

Handshake parameters related to low delay interface (sub frame level data communication) for output data of video decoder:

Parameter Name	**Details**	**Valid values**
tpp_H265_DataSyncDesc::nNumBlocks	Number of data blocks	Number of CTU rows given out by decoder in this call of fOutputLowDelayFxn. Each data block is having 16 lines of video. nNumBlocks = (k*nNumCTURows) where k = 1, 2, or 4 depending on CTU size (16x16, 32x32, or 64x64) repectively. Example: if nNumCTURows = 2 nNumBlocks = 2 (CTU size: 16x16) nNumBlocks = 4 (CTU size: 32x32) nNumBlocks = 8 (CTU size: 64x64)

If application wants to use video decoder to operate with low delay interface (sub frame) on output side

- It should create the video decoder with tpp_H265_CreateParams::nOutputLowDelayMode = PP_H265_NUMCTUROWS.
- It should set tpp_H265_DynamicParams::fOutputLowDelayFxn = non-NULL; to use low delay interface (sub frame level data communication).
- Address of the Luma and chroma output buffer will be present in decoded/display buffs. It will not be communicated via tpp_H265_DataSyncDesc structure.
- Constraint: display order not being same as decode order with tpp_H265_CreateParams::nOutputLowDelayMode = PP_H265_NUMCTUROWS, is an erroneous situation.
- tpp_H265_DynamicParams::fOutputLowDelayFxn == NULL && tpp_H265_CreateParams::nOutputLowDelayMode = PP_H265_NUMCTUROWS is an erroneous situation and codec returns error during vDecode() call.

B.3 Details of using Low Delay Interface at input side

This section explains the low delay interface details for input.

This feature can be enabled/disabled through create time parameters `tPP_H265_CreateParams:: nInputLowDelayMode`.

Creation time parameter related to sub frame level data communication for input data of video decoder:

Parameter Name	**Details**	**Valid values**	
tpp_H265_CreateParams::nInputLowDelayMode	Defines the mode of accepting the input data.	*PP_H265_ENTIREINPUTFRAME*	bit-stream provided to decoder is having entire frame
		PP_H265_SLICEMODE	bit-stream is provided to decoder after having a single(or more) number of slice NAL units

Dynamic parameters related to low delay interface (sub frame level data communication) for input data of video decoder:

Parameter Name	**Details**	**Valid values**
tpp_H265_DynamicParams:: fInputLowDelayFxn	This function is provided by the app/framework to the video decoder. The decoder calls this function to get partial compressed bit-stream data from the app/framework. App/framework that doesn't support datasync should set this to NULL.	*Any non-NULL value if nInputLowDelayMode != PP_H265_ENTIREINPUTFRAME*

Incase of nInputLow DelayMode = PP_H265_SLICEMODE, following points should be noticed

- No data is assumed to be available during vDecode() call except first vDecode() call, hence tPPInput_BitStream::nBufLength is don't care from second vDecode() call onwards. All the data has to be provided via data sync calls.

Handshake parameters related to low delay interface (sub frame level data communication) for input data of video decoder (nInputLowDelayMode = PP_H265_SLICEMODE:

Parameter Name	**Details**	**Valid values**
tpp_H265_DataSyncDesc::nSize	Size of the input data	Size of single slice (or) more number of slice NAL units.

If application wants to use video decoder to operate with low delay interface (sub frame on input side

- It should create the video decoder with tpp_H265_CreateParams::nInputLowDelayMode = PP_H265_SLICEMODE.
- It should set tpp_H265_DynamicParams::fInputLowDelayFxn = non-NULL; to use sub frame level data communication.
- tpp_H265_DynamicParams::fInputLowDelayFxn == NULL && tpp_H265_CreateParams::nInputLowDelayMode = PP_H265_SLICEMODE is an erroneous situation and codec returns error during vDecode() call