Chapter 1 Introduction

This chapter presents general information about C&M Video Codec IP

- Overview
- Features

1.1 Overview

The CodaDx6 is an official product name of C&M’s TRISTANeX video IP. Chips&Media’s CodaDx6 video codec IP is a high performance multi-standard video codec engine that can perform multiple standard encoding and decoding operations on a single video processing unit (VPU) such as H.263P3, MPEG-4 simple profile, and H.264 baseline profile codec. C&M’s video codec IP generally supports duplex codec operations up to full D1 (720x576) resolution in real-time, and especially it supports simultaneous multi-standard codec operations with multiple instances such as multi-party video conferencing, multi-tasking of video player and camcoder operations with different video coding standards.

Most of video engines in C&M’s video codec IP are optimally designed for shared usage between different video standards, which enables to achieve the ultra low power and low gate count with powerful performance. A simple block diagram of CodaDx6 video codec IP is presented in Figure 1-1. As shown in this figure, C&M’s all video codec IP has a 16-bit DSP core, Chips & Media’s proprietary BIT processor, which controls all the internal video codec operations. BIT processor also handles all communication between host processor and video codec IP through the host interface.

C&M’s video codec IP embedded in a customer-specific system-on-chip (SOC), will be controlled by host processor in SOC, and be used as VPU in SOC for handling all video related operations. For the simple and efficient control of this VPU by host processor, C&M video codec IP provides a set of registers called Host Interface Registers. All the commands and responses between host processor and VPU will be transmitted through Host Interface Registers. Stream data and some output picture data can be directly accessed by host processor, but all the commands and responses can only be exchanged through Host Interface Registers.

In order to provide more comprehensive way of controlling video codec IP, C&M also provides a set of API functions which fully covers all the required operations from host processor side.

CodaDx6 also has a 16-bit DSP core, Chips & Media’s proprietary BIT processor. BIT processor controls all the internal video decoder operations, and the resulting computational power on external host processor can be reduced noticeably. BIT processor also handles communication through the host interface. By using some dedicated registers, host can control CodaDx6 video IP while maintaining the low computational cost.
1.2 Basic Features

The main features of CodaDx6 are fully compliant with MPEG-4 SP, H.263 P3 and H.264 BP. The image size up to full D1 (720x576 or 720x480) is supported in both encoding and decoding. The CodaDx6 can support various error resilience tools and very effective embedded rate control scheme. So the CodaDx6 is easy to integrate into the system because its interface is composed of general and simple AXI/APB. And the CodaDx6 provides programmability, flexibility and easy of upgrade in decoding or host interface because all the controls in decoding process and host interface are implemented as firmware in a programmable BIT processor.

Detailed features of C&M’s Video Codec IP can be summarized as follows:

a) Multi-standard video codec
   - MPEG-4 part-II simple profile encoding and decoding
   - H.264/AVC baseline profile encoding and decoding
   - H.263 P3 encoding and decoding

b) Multi-instance video codec
   - CodaDx6 can decode an MPEG-4 bitstream and a H.264 bitstream simultaneously.
• CodaDx6 can encode an MPEG-4 bitstream and decode a H.264 bitstream simultaneously.

c) Encoding tools
• [-16, +6] 1/2 and 1/4-pel accuracy motion estimation.
• 16x16, 16x8, 8x16, 8x8 block sizes are supported.
  — Available block sizes can be configurable.
  — The encoder uses only one reference frame for the motion estimation
• Unrestricted motion vector.
• Prediction
  — MPEG-4 AC/DC prediction
  — H.264/AVC intr-prediction
  — H.263 Annex J, K(RS=0 and ASO=0), and T are supported
• Error resilience tools
  — MPEG-4 resync-marker & data-partitioning with RVLC which supports two different packetization rules (number of bits or number of MB’s)
• CIR (Cyclic Intra Refresh)/AIR (Adaptive Intra Refresh)
• Bit-rate control (CBR & VBR)
• Minimum encoding size is 32 pixels in horizontal and 16 pixels in vertical.

d) Decoding tools
• H.264 Baseline Profile
  — Supports all H.264 baseline profile features
  — Variable block size ( 16x16, 16x8, 8x16, 8x8, 8x4, 4x8 and 4x4)
  — Error detection, concealment and error resilience tools such as FMO, ASO etc.
• MPEG-4 Simple Profile
  — Supports all MPEG-4 simple profile features
  — 4MV with unrestricted motion vector
  — Error resilience tools: re-sync marker, data-partitioning with RVLC
• H.263 P3
  — Baseline profile plus Annex I, J, K (RS=0 and ASO=0), and T are supported.
• Minimum decoding size is 16 pixels in horizontal and 16 pixels in vertical.

e) Pre/Post rotation & Mirroring
• 90 x n (n=0,1,2,3) degree rotation /w mirroring for incoming image
• 90 x n (n=0,1,2,3) degree rotation /w mirroring for decoded image

f) Programmability
• The CodaDx6 embeds C&M proprietary 16-bit DSP processor dedicated to processing bitstream and controlling operation of the decoder hardware.
• The CodaDx6 provides a set of general purpose registers and interrupts for communication between a host processor and the video IP.
g) Performance

- All video standard decoders support up to 720x480 @ 30 fps and 720x576 @ 25 fps.
- All video standard encoders support up to 720x480 @ 30 fps and 720x576 @ 25 fps.
- CodaDx6 supports full duplex operations up to VGA(640x480) @ 30 fps.
- CodaDx6 supports half duplex operations up to 720x480 @ 30 fps and 720x576 @ 25 fps.
Chapter 2  Host Interface

This chapter describes the details of the host interfaces provided by C&M’s video codec IP.

- Overview
- Host Interface Registers

2.1  Overview of Host Interface

This section presents the general description of host interfaces provided for host processor to control C&M’s video codec IP.

2.1.1  Communication Models

Generally speaking, VPU requires a path for exchanging data and/or messages between host processor and VPU itself. C&M’s VPU assumes that a shared external memory for data transmission is provided from host through ABMA host bus (AHB or AXI), and control path for setting some internal registers on VPU is also provided through AMBA peripheral bus. So based on this assumption, host should manage separate paths for data and control. A detailed diagram illustrating this basic assumption is presented in Figure 2.1.

Figure 2-1. Exchanging Data and Messages between Host and VPU

All the bitstream data and picture data will be directly accessed by host processor and VPU. And the related information about all those data transfer will be exchanged through host interface as well as command and responses. In order to do this, host interface of VPU reserves a set of registers accessible from host processor. Some of these host registers will be used for exchanging actual command and
responses. And other registers will be used to give some information about internal status of VPU to host processor.

Basically, C&M provides a set of pre-defined commands and their corresponding responses. Firmware running on BIT processor has been well-optimized for these given set of commands and responses. Even though all general commands and responses can be handled by this set, some customer-specific commands and responses could be easily supported by changing the definition of interface registers, as well as part of BIT processor code.

### 2.1.2 Host Interface Registers

C&M provides a set of commands for controlling encoder and decoder operations with a frame-by-frame basis as well as corresponding responses. These host interface registers can be partitioned into three categories as follows:

- **BIT Processor Control Registers:**
  
  Host interface registers in this category will be used to update or show BIT processor status to host processors. Most of these registers will be used for initializing BIT processor during boot-up.

- **BIT Processor Global Registers:**
  
  Host interface registers in this category will be used to store all the global variables which will be kept even while active instance is changed. Basically all the buffer addresses and some global options will be safely stored into these registers.

- **BIT Processor Command I/O Registers:**
  
  Host interface registers in this category will be overwritten or updated whenever new command is transmitted from Host processor. All the commands with input arguments and all the corresponding responses with return values will be handled by using these registers.

Among them, Command I/O Registers will be used by a pre-defined way for each command for controlling VPU.

### 2.1.3 Data Handling

All the transactions handling some output pixel data or stream data will be directly performed by host processor or VPU itself through shared memory space on external SDRAM. In order to assure safe transactions between host processor and VPU, all the required information will be stored into host interface registers and will also be referred anytime by host processor or VPU. Generally, all of these transactions are one-directional transactions, which means the one only writes data and the other only reads the written data on a single data buffer like stream buffer case. So by using a pair of read pointer and write pointer, all those transactions can be controlled very easily and safely.

As well as common data buffers on shared memory, BIT processor requires a certain amount of memory for processing video encoder and decoder, which is called as “Working Buffer”. This BIT processor Working Buffer could only be accessible by VPU. As well as this Working Buffer, frame buffers used in

---

**API&Software Programmer’s User Guide**

Chips&Media, Freescale Semiconductor
picture encoding and decoding will be managed by VPU only, which will prevent abnormal access from host applications and ensure safe encoding and decoding in VPU.

For proper streaming, available free space in decoder stream buffer and available amount of new stream data in encoder stream buffer can be referred by using the buffer read pointer, write pointer and buffer size. C&M provides a set of API’s for this purpose, which can be called anytime application wants to call.

### 2.2 API-based VPU Control

Host application generally controls VPU through pre-defined API set by sending a command and arguments, and after interrupt signaling for completion of requested operation, host application can get the results from API functions as shown in Figure 2.2.

**Figure 2-2. SW Control Model of VPU from Host Application**

Each API definition includes the requested command as well as input data structure and output data structure. The given command from API function will always be written on a dedicated I/O register, but the input data structure and output data structure will be transmitted through a set of command I/O registers which contains input arguments and output results. So application developers do not need to struggle with host register definitions and their usage.
Chapter 3  VPU Driver API Reference

3.1  API Features

In order to provide an efficient way of controlling VPU based on C&M’s video codec API, C&M provides a set of API functions, VPU API, which fully covers functionalities of C&M video codec IP. This VPU API-based approach will be very helpful to speed up development process of application SW. Important features of API sets from C&M can be summarized as follows:

3.1.1  Simple S/W control

C&M’s VPU API provides the simplest way to control video codec IP and the easiest way to avoid malfunctions in application SW. Application does not need to know all the details of video IP’s internal operations which is irrelevant to application developers. For example, in order to initialize video codec IP, application just calls an API for initialization vpu_Init(), and all the information required for calling this API is the physical address of working buffer. vpu_Init() API performs all the required steps for initializing C&M’s video IP. When issuing picture encoder and decoder operation, application just need to change some updated variables included in the well-defined input data structure.

3.1.2  Handling Multi-instances

C&M’s video codec IP supports multiple instances for encoding and decoding at the same time, which will be crucial for supporting multiple video encoder or decoder tasks such as video conferencing with multi-party call.

In order to support this multi-instance operations, C&M’s VPU API provides full set of functions required for handling instances easily. When opening a new instance, application will get a handle specifying the new instance if a new handle is available at that time. And all the following operations for a given new instance could be separately controlled by using the received handle from VPU API.

3.1.3  Frame-based Codec Processing

C&M’s video codec IP runs encoding and decoding on a frame-by-frame basis, which enables low level dependency of VPU operations on host processor. While frame processing (i.e. encoding or decoding) is running, there is no need for exchanging command and responses between host processor and VPU. This is the key feature for promising the lowest burden to host processor while providing easy way of controlling each video codec instance.

3.2  Type Definitions

This section describes the list of types and structures of used in the VPU API, which includes the following contents:

•  Type Definitions
•  Data Structure Definitions
3.2.1 Type Definitions

This subsection describes the common data types used in VPU API functions.

3.2.1.1 Uint8
typedef unsigned char Uint8;

Description:
This type is an 8-bit unsigned integral type, which is used for declaring pixel data.

3.2.1.2 Uint16
typedef unsigned short Uint32;

Description:
This type is a 16-bit unsigned integral type.

3.2.1.3 Uint32
typedef unsigned int Uint32;

Description:
This type is a 32-bit unsigned integral type, which is used for declaring variables with wide ranges and no signs such as size of buffer.

3.2.1.4 PhysicalAddress
typedef Uint32 PhysicalAddress;

Description:
This is a type for representing physical addresses which is recognizable by VPU module. In general, VPU hardware does not know about virtual address space which is set and handled by host processor. All these virtual addresses are translated into physical addresses by Memory Management Unit. All data buffer addresses such as stream buffer, frame buffer, should be given to VPU module as an address on physical address space.

3.2.1.5 CodStd
typedef enum {
STD_MPEG4 = 0,
STD_H263,
STD_AVC
} CodStd;
**Description:**

This is an enumeration for declaring code standard type variables. Currently, four different video standards, MPEG4 SP, H.263P3, AVC (H.264) BP codecs are supported by VPU module.

### 3.2.1.6 RetCode

typedef enum {
    RETCODE_SUCCESS,
    RETCODE_FAILURE,
    RETCODE_INVALID_HANDLE,
    RETCODE_INVALID_PARAM,
    RETCODE_INVALID_COMMAND,
    RETCODE_ROTATOR_OUTPUT_NOT_SET,
    RETCODE_ROTATOR_STRIDE_NOT_SET,
    RETCODE_FRAME_NOT_COMPLETE,
    RETCODE_INVALID_FRAME_BUFFER,
    RETCODE_INSUFFICIENT_FRAME_BUFFERS,
    RETCODE_INVALID_STRIDE,
    RETCODE_WRONG_CALL_SEQUENCE,
    RETCODE_CALLED_BEFORE,
    RETCODE_NOT_INITIALIZED,
    RETCODE_DEBLOCKING_OUTPUT_NOT_SET,
    RETCODE_FAILURE_TIMEOUT,
    RETCODE_BUSY,
    RETCODE_IDLE
} RetCode;

**Description:**

This is an enumeration for declaring return codes from API function calls. The meaning of each return code is the same for all API functions, but the reason of non-successful return might be different. Some details of those reasons will be briefly described in API definition chapter. In this chapter, the basic meaning of each return code will be presented.

**RETCODE_SUCCESS:**

This means operation was done successfully.
RETCODE_FAILURE:
This means operation was not done successfully.

RETCODE_INVALID_HANDLE:
This means the given handle for current API function call was invalid (for example, not initialized yet, improper function call for the given handle, etc.).

RETCODE_INVALID_PARAM:
This means the given argument parameters (for example, input data structure) was invalid (not initialized yet or not valid anymore).

RETCODE_INVALID_COMMAND:
This means the given command was invalid (for example, undefined, or not allowed in the given instances).

RETCODE_ROTATOR_OUTPUT_NOT_SET:
This means rotator output buffer is not allocated even though rotation is enabled.

RETCODE_ROTATOR_STRIDE_NOT_SET:
This means rotator stride is not provided even though rotation is enabled.

RETCODE_FRAME_NOT_COMPLETE:
This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be allowed.

RETCODE_INVALID_FRAME_BUFFER:
This means certain frame buffer pointers were invalid (not initialized yet or not valid anymore).

RETCODE_INSUFFICIENT_FRAME_BUFFERS:
This means the given numbers of frame buffers were not enough for the operations of the given handle. This return code will only be received when calling vpu_Enc/DecRegisterFrameBuffer() functions.

RETCODE_INVALID_STRIDE:
This means the given stride was invalid (for example, 0, not a multiple of 8 or smaller than picture size). This return code is only allowed in API functions which set stride.

RETCODE_WRONG_CALL_SEQUENCE:
This means current API function call was invalid considering the allowed sequences between API functions (for example, missing one crucial function call before this function call).

RETCODE_CALLED_BEFORE:
This means multiple calls of current API function for a given instance are invalid.

RETCODE_NOT_INITIALIZED:
This means VPU module was not initialized yet. Before calling any API functions, the initialization API function, `vpu_Init()`, should be called at the beginning.

**RETCODE_DEBLOCKING_OUTPUT_NOT_SET:**

This means deblocking output information was not available even though deblocking filter had been activated by application. If application wants to enable deblocking filter option in MPEG-4 case, application should register the frame buffer information of deblocking filtered output by using `vpu_DecGiveCommand()`. If not, application will get this return code when calling `vpu_DecStartOneFrame()`.

**RETCODE_FAILURE_TIMEOUT:**

This means the calling on API is timeout.

**RETCODE_BUSY:**

The return value of API function `vpu_IsBusy()`, it means the vpu HW is busy.

**RETCODE_IDLE:**

The return value of API function `vpu_IsBusy`, it means the vpu HW is idle.

### 3.2.1.7 CodecCommand

typedef enum {
    ENABLE_ROTATION,
    DISABLE_ROTATION,
    ENABLE_MIRRORING,
    DISABLE_MIRRORING,
    SET_MIRROR_DIRECTION,
    SET_ROTATION_ANGLE,
    SET_ROTATOR_OUTPUT,
    SET_ROTATOR_STRIDE,
    ENC_GET_SPS_RBSP,
    ENC_GET_PPS_RBSP,
    DEC_SET_SPS_RBSP,
    DEC_SET_PPS_RBSP,
    ENC_PUT_MP4_HEADER,
    ENC_PUT_AVC_HEADER,
}
ENC_SET_SEARCHRAM_PARAM,
ENC_GET_VOS_HEADER,
ENC_GET_VO_HEADER,
ENC_GET_VOL_HEADER,
ENC_SET_INTRA_MB_REFRESH_NUMBER,
ENC_ENABLE_HEC,
ENC_DISABLE_HEC,
ENC_SET_SLICE_INFO,
ENC_SET_GOP_NUMBER,
ENC_SET_INTRA_QP,
ENC_SET_BITRATE,
ENC_SET_FRAME_RATE

} CodecCommand;

Description:

This is a special enumeration type for some configuration commands from host processor to VPU module. Most of these commands can be called occasionally, not periodically for changing the configuration of codec operations running on VPU module. Details of these commands will be presented in the next subsection.

3.2.1.8 MirrorDirection

typedef enum {

    MIRDIR_NONE,
    MIRDIR_VER,
    MIRDIR_HOR,
    MIRDIR_HOR_VER

} MirrorDirection;

Description:

This is an enumeration type for representing the mirroring direction.

3.2.1.9 Mp4HeaderType

typedef enum {

    VOL_HEADER, // video object layer header


VOS_HEADER, // visual object sequence header
VIS_HEADER // video object header
} Mp4HeaderType;

**Description:**
This is a special enumeration type for MPEG-4 top-level header classes such as visual sequence header, visual object header and video object layer header.

### 3.2.1.10 AvcHeaderType
typedef enum {
    SPS_RBSP, // sequence parameter set
    PPS_RBSP // picture parameter set
} AvcHeaderType;

**Description:**
This is a special enumeration type for AVC parameter sets required in encoder and decoder such as sequence parameter set and picture parameter set.

### 3.2.1.11 EncHandle
typedef EncInst * EncHandle;

**Description:**
This is a dedicated type for encoder handle returned when an encoder instance is opened. An encoder instance can be referred by the corresponding handle. EncInst is a type managed internally by API. Application does not need to care about it.

### 3.2.1.12 DecHandle
typedef DecInst * DecHandle;

**Description:**
This is a dedicated type for decoder handle returned when a decoder instance is opened. A decoder instance can be referred by the corresponding handle. DecInst is a type managed internally by API. Application does not need to care about it.

### 3.2.2 Data & Structure Definitions
This section describes data and structure definitions used in VPU API functions.

### 3.2.2.1 FrameBuffer
typedef struct {

PhysicalAddress bufY;
PhysicalAddress bufCb;
PhysicalAddress bufCr;

} FrameBuffer;

**Description:**
This is a data structure for representing frame buffer pointers of each color components.

*bufY*
Address for Y component in the physical address space.

*bufCb*
Address for Cb component in the physical address space.

*bufCr*
Address for Cr component in the physical address space. All of the 3 addresses must be 4-byte aligned. One pixel value of a component occupies one byte, and the frame data is in YCbCr 4:2:0-format. So the sizes of Cb and Cr buffer are 1/4 of Y buffer size. Host application must allocate external SDRAM spaces for those components by using this data structure.

### 3.2.2.2 RECT

typedef struct {
    int left;
    int top;
    int right;
    int bottom;
} RECT;

**Description:**
This is a data structure for representing rectangular window information in a frame.

### 3.2.2.3 EncHeaderParam

typedef struct {
    PhysicalAddress PhysBuf;
    Uint8 *VirtBuf;
    int size;
}
int headerType;
} EncHeaderParam;

**Description:**
This structure is used for adding a header syntax layer into the encoded bit stream. Parameter *headerType* is the input parameter to VPU module, and the other three parameters are returned value from VPU module after completing requested operation.

### 3.2.2.4 EncParamSet

typedef struct {
    Uint8 *paramSet;
    int size;
} EncParamSet;

**Description:**
This structure is used when host processor additionally wants to get SPS data or PPS data from encoder instance. The resulting SPS data or PPS data can be used in real application as a kind of “out-of-band” information.

### 3.2.2.5 EncMp4Param

typedef struct {
    int mp4_dataPartitionEnable;
    int mp4_reversibleVlcEnable;
    int mp4_intraDcVlcThr;
    int mp4_hecEnable;
    int mp4_verid;
} EncMp4Param;

**Description:**
This is a data structure for configuring MPEG4-specific parameters in encoder applications

*mp4_dataPartitionEnable*

0 – disable, 1 – enable

*mp4_reversibleVlcEnable*

0 – disable, 1 – enable

*mp4_intraDcVlcThr*

setting intra_dc_vlc_thr in MPEG-4 part 2 standard. Valid range is 0 ~ 7.
mp4_hecEnable
   0 - disable, 1 - enable

mp4_verid
   Value of MPEG-4 part 2 standard version ID. Version 1 and version 2 are allowed.

3.2.2.6  EncH263Param
typedef struct {
   int h263_annexJEnable;
   int h263_annexKEnable;
   int h263_annexTEnable;
} EncH263Param;

Description:
This is a data structure for configuring H.263-specific parameters in encoder applications

h263_annexJEnable
   0 – disable, 1 – enable

h263_annexKEnable
   0 – disable, 1 – enable

h263_annexTEnable
   0 – disable, 1 – enable

3.2.2.7  EncAvcParam
typedef struct {
   int avc_constrainedIntraPredFlag;
   int avc_disableDeblk;
   int avc_deblkFilterOffsetAlpha;
   int avc_deblkFilterOffsetBeta;
   int avc_chromaQpOffset;
   int avc_audEnable;
   int avc_fmoEnable;
   int avc_fmoSliceNum;
   int avc_fmoType;
EncAvcParam;

**Description:**
This is a data structure for configuring AVC-specific parameters in encoder applications.

- **avc_constrainedIntraPredFlag**
  - 0 – disable, 1 – enable

- **avc_disableDeblk**
  - 0 – enable, 1 - disable, 2 - disable deblocking filter at slice boundaries

- **avc_deblkFilterOffsetAlpha**
  - deblk_filter_offset_alpha (-6 ~ 6)

- **avc_deblkFilterOffsetBeta**
  - deblk_filter_offset_beta (-6 ~ 6)

- **avc_chromaQpOffset**
  - chroma_qp_offset (-12 ~ 12)

- **avc_audEnable**
  - 0 – disable, 1 – enable
  - If this is 1, the encoder will generate AUD RBSP at the start of every picture.

- **avc_fmoEnable**
  - 0 – disable, 1 – enable
  - If this is 1, the encoder will enable FMO option and the following two variables should be set by proper numbers.

- **avc_fmoSliceNum**
  - Number of slice groups in a picture. This value should be in a range between 2 and 8.

- **avc_fmoType**
  - 0 – interleaved, 1 – dispersed
  - This is a variable for representing FMO type. We supported two most popular FMO types, interleaved and dispersed. Other options will not be allowed.

### 3.2.2.8 EncSliceMode

typedef struct {
  int sliceMode;
  int sliceSizeMode;
}
int sliceSize;

} EncSliceMode;

**Description:**

This structure is used for declaring encoder slice mode and its options.

*sliceMode*

0 - One slice per picture, 1- Multiple slices per picture.

In normal MPEG-4 mode, resync-marker and packet header are inserted between slice boundaries. In short video header with Annex K = 0, GOB header is inserted at every GOB layer start. In short video header with Annex K = 1, multiple slices are generated. In AVC mode, multiple slice layer RBSP is generated.

*sliceSizeMode*

This parameter means the size of a generated slice when sliceMode = 1, 0 means sliceSize is define by amount of bits, and 1 means sliceSize is defined by the number of MB's in a slice. This parameter will be ignored when sliceMode = 0 or in short video header mode with Annex K = 0.

*sliceSize*

Size of a slice in bits or in MB numbers included in a slice, which will be specified by the variable, sliceSizeMode. This parameter will be ignored when sliceMode = 0 or in short video header mode with Annex K = 0.

---

### 3.2.2.9 EncOpenParam

typedef struct {
    
    CodStd bitstreamFormat;
    PhysicalAddress bitstreamBuffer;
    Uint32 bitstreamBufferSize;
    Uint8 *virt_bitstreamBuffer;
    int picWidth;
    int picHeight;
    Uint32 frameRateInfo;

} EncOpenParam;
int bitRate;
int initialDelay;
int vbvBufferSize;
int enableAutoSkip;
int gopSize;
EncSliceMode slicemode;
int intraRefresh;
int sliceReport;
int mbReport;
int mbQpReport;
int rcIntraQp;
int dynamicAllocEnable;
int ringBufferEnable;
union {
  EncMp4Param mp4Param;
  EncH263Param h263Param;
  EncAvcParam avcParam;
} EncStdParam;
}
}
EncOpenParam;

Description:
Data structure for parameters when an encoder instance is opened.

bitstreamFormat
Standard type of bitstream in encoder operation. It will be one of STD_MPEG4, STD_H263, and
STD_AVC.

bitstreamBuffer
Start physical address of bit stream buffer into which encoder put bit streams. This address must be a
multiple of 512, namely, 512 byte-aligned.

bitstreamBufferSize
Size in bytes of a buffer pointed to by bitstreamBuffer. This value must be a multiple of 1024. The
maximum size is 16383 x 1024 bytes.

virt_bitstreamBuffer
Start virtual address of bit stream buffer into which encoder put bit streams. This address must be a multiple of 512, namely, 512 byte-aligned.

\textit{picWidth}

Width of a picture to be encoded in pixels.

\textit{picHeight}

Height of a picture to be encoded in pixels.

\textit{frameRateInfo}

The 16 LSB bits, \([15:0]\), is a numerator and 16 MSB bits, \([31:16]\), is a denominator for calculating frame rate. The numerator means clock ticks per second, and the denominator is clock ticks between frames minus 1. So the frame rate can be defined by \((\text{numerator}/(\text{denominator} + 1))\), which equals to \((\text{frameRateInfo} \& \text{0xffff})/((\text{frameRateInfo} >> 16) + 1)\).

For example, the value 30 of frameRateInfo represents 30 frames/sec, and the value 0x3e87530 represents 29.97 frames/sec.

\textit{bitRate}

Target bit rate in kbps. If 0, there will be no rate control, and pictures will be encoded with a quantization parameter equal to quantParam in EncParam.

\textit{initialDelay}

Time delay (in mili-seconds) it takes for the bit stream to reach initial occupancy of the vbv buffer from zero level. This value is ignored if rate control is disabled. The value 0 means the encoder does not check for reference decoder buffer delay constraints.

\textit{vbvBufferSize}

\text{vbv.buffer.size} in bits. This value is ignored if rate control is disabled or \textit{initialDelay} is 0. The value 0 means the encoder does not check for reference decoder buffer size constraints.

\textit{enableAutoSkip}

The value 0 disables automatic skip and 1 enables automatic skip in encoder operation. Automatic skip means encoder can skip frame encoding when generated Bitstream so far is too big considering target bitrate. This parameter will be ignored if rate control is not used (bitRate = 0).

\textit{gopSize}

GOP size

0 – Only first picture is I, 1 – all I pictures, 2 – IPIP…, 3 – IPPIPP…

The maximum value is limited by 60.

\textit{slicemode}

This structure indicates encoder slice mode and its options.

\textit{intraRefresh}
0 – Intra MB refresh is not used.
Otherwise – At least N MB’s in every P-frame will be encoded as intra MB’s.

sliceReport
If this value is 1, the encoder will generate the end address + 1 of every slice in a picture. If 0, the feature is not used. For detail, see sliceInfo in EncOutputInfo.

mbReport
If this value is 1, the encoder will generate the bit position every macroblock in a picture. If 0, the feature is not used. For detail, see mbInfo in EncOutputInfo.

mbQpReport
0 – disable, 1 – enable

When this field is set in MPEG4 and H.263 case, encoder will provide MB Qstep values as output

rcIntraQp
Quantization parameter for I frame. When this value is -1, quantization parameter for I frames is automatically determined by VPU.

dynamicBuffAllocEnable
0 - disable, 1 - enable

When this field is set, dynamic buffer allocation is enabled under buffer reset mode for encoder operation, which means buffer start address specified during ENC_SEQ_INIT, bitstreamBuffer, will be ignored in picture encoding operation. In this case, picture buffer start address should be specified at every call of ENC_PIC_RUN command.

When this field is not set, picture buffer start address given by bitstreamBuffer, will be used in encoder operation, even though buffer reset mode is enabled.

ringBufferEnable
0 - disable, 1 - enable

This flag declares the streaming mode for current encoder instance. Two streaming modes, packet-based streaming with ring-buffer (buffer-reset mode) and frame-based streaming with line buffer (buffer-flush mode), can be configured by using this flag.
When this field is set, packet-based streaming with ring-buffer will be used. And when this field is not set, frame-based streaming with line-buffer will be used.

informative information.

**mp4Param**
Parameters for MPEG-4 part 2 Visual.

**h263Param**
Parameters for ITU-T H.263.

**avcParam**
Parameters for AVC.

### 3.2.2.10 EncInitialInfo

typedef struct {
    int minFrameBufferCount;
} EncInitialInfo;

**Description:**
This is a data structure for parameters of `vpu_EncGetInitialInfo()` which required to get the minimum required buffer count in host applications. This returned value will be used to allocate frame buffers in `vpu_EncRegisterFrameBuffer()`.

### 3.2.2.11 EncParam

typedef struct {
    FrameBuffer * sourceFrame;
    int forceIPicture;
    int skipPicture;
    int quantParam;
    PhysicalAddress picStreamBufferAddr;
    int picStreamBufferSize;
    int intraRefresh;
    int hecEnable;
    EncSliceMode slicemode;
}
} EncParam;

**Description:**

This is a data structure for configuring one frame encoding operation.

*sourceFrame*

This member must represent the frame buffer containing source image to be encoded.

*forceIPicture*

If this value is 0, the picture type is determined by VPU module according to the various parameters such as encoded frame number and GOP size.

If this value is 1, the frame will be encoded as an I-picture regardless of the frame number or GOP size, and I-picture period calculation will be reset to initial state. In MPEG-4 and H.263 case, I-picture is sufficient for decoder refresh. In H.264 case, the picture will be encoded as an IDR (Instantaneous Decoding Refresh) picture. This value will be ignored if skipPicture = 1.

*skipPicture*

If this value is 0, the encoder encodes the picture as normal.

If this value is 1, the encoder ignores sourceFrame and generates a skipped picture. In this case, the reconstructed image at decoder side is a duplication of the previous picture. The skipped picture is encoded as P-type regardless of GOP size.

*quantParam*

This value will be used for all quantization parameters in case of VBR(no rate control). The range of value will be 1 ~ 31 for MPEG-4, 0 ~ 51 for H.264. When rate control is enabled, this field is ignored.

*picStreamBufferAddr*

The start address of a picture stream buffer under line-buffer mode and dynamic buffer allocation.

This variable represents the start of a picture stream for encoded output. In buffer-reset mode, application might use multiple picture stream buffers for the best performance. By using this variable, application could re-register the start position of the picture stream while issuing a picture encoding operation. This start address of this buffer must be 4-byte aligned, and its size will be specified the following variable, picStreamBufferSize. In packet-based streaming with ring-buffer, this variable will be ignored.

Note: This variable is only meaningful when both line-buffer mode and dynamic buffer allocation are enabled.

*picStreamBufferSize*

Byte size of a picture stream chunk
This variable represents byte size of a picture stream buffer. This variable is so crucial in line-buffer mode. That's because encoder output could be corrupted if this size is smaller than any picture encoded output. So this value should be big enough for storing multiple picture streams with average size. In packet-based streaming with ring-buffer, this variable will be ignored.

Note: This variable will specify the picture stream buffer size for encoded output under line-buffer mode.

\textit{intraRefresh}

- 0 – Intra MB refresh is not used.
- Otherwise – At least \( N \) MB’s in every P-frame will be encoded as intra MB’s.

\textit{hecEnable}

- 0 - disable, 1 - enable

\textit{slicemode}

This structure indicates encoder slice mode and its options.

### 3.2.2.12 EncOutputInfo

typedef struct {
    PhysicalAddress bitstreamBuffer;
    Uint32 bitstreamSize;
    int bitstramWrapAround;
    int picType;
        int numOfSlices;
    Uint8 *sliceInfo;
    Uint8 *mbInfo;
    Uint8 *mbQpInfo;
} EncOutputInfo;

\textbf{Description:}

This is a data structure for reporting the results of picture encoding operations.

\textit{bitstreamBuffer}

Physical address of the starting point of newly encoded picture stream.
bitstreamSize

Byte size of encoded bitstream.

bitstramWrapAround

Indicate whether BitStream buffer wrap is arounded or not. 1 - Bitstream buffer is arounded and 0 - Not arounded.

picType

Coded picture type

In H.263 and MPEG4, 0 - I picture, and 1 - P picture.

In AVC, 0 – IDR picture and 1 – Non-IDR picture

numOfSlices

Number of slices included in newly encoded picture.

When sliceReport in EncOpenParam is 0, this value is invalid.

sliceInfo

Virtual address of data array storing each slice end position.

This is a set of relative position of each slice end from the starting point of the last encoded picture.

If application sets sliceReport = 0 when opening an encoder instance, this value is invalid.

mbInfo

Virtual address of data array storing start position of each macroblock The start positions are relative positions from the beginning of newly encoded bitstream. If application sets mbReport = 0, this value is invalid.

mbQpInfo

Base virtual address of MB Qstep values with a pre-defined format.

3.2.2.13 DecParamSet

typedef struct {
    Uint8 * paraSet;
    int size;
} DecParamSet;

Description:

This structure is used when host processor additionally wants to send SPS data or PPS data from external way. The resulting SPS data or PPS data can be used in real applications as a kind of “out-of-band” information.
3.2.2.14 DecOpenParam

typedef struct {
    CodStd bitstreamFormat;
    PhysicalAddress bitstreamBuffer;
    Uint8 *virt_bitstreamBuffer;
    int bitstreamBufferSize;
    int qpReport;
    int reorderEnable;
    int filePlayEnable;
    int picWidth;
    int picHeight;
    int streamStartByteOffset;
    PhysicalAddress psSaveBuffer;
    int psSaveBufferSize;
} DecOpenParam;

Description:
This data structure is used when host application wants to open a new decoder instance.

bitstreamFormat
Standard type of bitstream in decoder operation. It will be one of STD_MPEG4, STD_H263, STD_AVC and STD_VC1.

bitstreamBuffer
Start physical address of bit stream buffer from which the decoder can get the next bitstream. This address must be 512 byte-aligned.

virt_bitstreamBuffer
Start virtual address of bit stream buffer from which the decoder can get the next bitstream. This address must be 512 byte-aligned.

bitstreamBufferSize
Size in bytes of a buffer pointed by bitstreamBuffer. This value must be a multiple of 1024. The maximum size is 16383 x 1024 bytes.

reorderEnable
The value 1 enables display buffer reordering when decoding H.264 streams. In H.264 case output decoded picture may be re-ordered if pic_order_cnt_type is “0” or “1.” In that case, decoder must
delay output display for re-ordering but some applications (ex. video telephony) don’t want such display delay. Host may set this flag to “0” to disable output display buffer reordering. Then BIT processor does not re-order output buffer when pic_order_cnt_type is “0” or “1.” If pic_order_cnt_type is “2” or MPEG4/H.263 case, this flag is ignored because output display buffer reordering is not allowed.

*filePlayEnable*

0 – disable, 1 – enable

When this field is set, file play mode will be enabled for decoder operation. File play mode means application should provide chunk size and reset write pointer at every frame processing.

*picWidth*

This variable represents the horizontal picture size read from file format header.

Note: This variable is not valid in current release, and added for future use. Setting this variable will not cause any changes in decoder operation.

*picHeight*

This variable represents the vertical picture size read from file format header.

Note: This variable is not valid in current release, and added for future use. Setting this variable will not cause any changes in decoder operation.

*streamStartByteOffset*

Start byte offset of the stream buffer. Since VPU has an internal limitation that stream buffer start address must be 4 byte-aligned, sometimes host application needs to copy the stream data to a separate 4-byte aligned buffer. By using this offset, this kind of overhead could be saved.

Note: This offset should be ranged between 0 and 3.

*psSaveBuffer*

Start address of PS (SPS/PPS) save buffer which the decoder can save PS (SPS/PPS) RBSP. This address must be 4 byte-aligned.

Note: This variable is only valid for H.264 decoder case

*psSaveBufferSize*

Size in bytes of a buffer pointed by psSaveBuffer

This value must be a multiple of 1024. The maximum size is 65565 x 1024 bytes.
Note: This variable is only valid when decodes H.264 stream.

### 3.2.2.15 DecInitialInfo

typedef struct {
    int picWidth;
    int picHeight;
    Uint32 frameRateInfo;
    Uint32 picCropEnable;
    int mp4_dataPartitionEnable;
    int mp4_reversibleVlcEnable;
    int mp4_shortVideoHeader;
    int h263_annexJEnable;
    int minFrameBufferCount;
    int frameBufDelay;
    int nextDecodedIdxNum;
    int normalSliceSize;
    int worstSliceSize;
} DecInitialInfo;

**Description:**
Data structure to get information necessary to start decoding from the decoder.

**picWidth**
Horizontal picture size in pixels.

This width value can be used while allocating decoder frame buffers. So this returned value would be bigger than the display picture size declared on stream header in MPEG-4 and VC1 cases. The actual returned picWidth value will be calculated from the declared display size as follows:

\[
\text{picWidth} = \left(\frac{\text{picXsize} + 15}{16}\right) \times 16,
\]

where picXsize is the horizontal picture size getting from stream. When picXsize is a multiple of 16, picWidth = picXsize.

**picHeight**
Vertical picture size in pixels.
This height value can be used while allocating decoder frame buffers. So this returned value would be bigger than the display picture size declared on stream header in MPEG-4 and VC1 cases. The actual returned picHeight value will be calculated from the declared display size as follows:

\[
\text{picHeight} = \left\{ \frac{\text{picYsize} + 15}{16}\right\} \times 16,
\]

where picYsize is the horizontal picture size getting from stream. When picYsize is a multiple of 16, picHeight = picYsize.

**frameRateInfo**

The 16 LSB bits, [15:0], is a numerator and 16 MSB bits, [31:16], is a denominator for calculating frame rate. The numerator means clock ticks per second, and the denominator is clock ticks between frames minus 1. So the frame rate can be defined by \((\text{numerator}/(\text{denominator} + 1))\), which equals to \((\text{frameRateInfo} \& 0xffff) /((\text{frameRateInfo} >> 16) + 1)\).

For example, the value 30 of frameRateInfo represents 30 frames/sec, and the value 0x3e87530 represents 29.97 frames/sec.

**picCropEnable**

0 – disable, 1 – enable

**mp4_dataPartitionEnable**

0 – disable, 1 – enable

**mp4_reversibleVlcEnable**

0 – disable, 1 – enable

**mp4_shortVideoHeader**

0 – disable, 1 – enable

**H263_annexJEnable**

0 – disable, 1 – enable

**minFrameBufferCount**

The minimum number of frame buffers required for decoding. Application must allocate at least this number of frame buffers and register those number of buffers to VPU module using vpu_DecRegisterFrameBuffer() before decoding pictures.

**frameBufDelay**

Maximum display frame buffer delay for buffering decoded picture reorder. VPU may delay decoded picture display for display reordering when H.264, pic_order_cnt_type “0” or “1” case (By default, some H.264 encoder set pic_order_cnt_type by 0 or 1, but in BP applications, this setting is not actually used in practice).

**nextDecodedIdxNum**

Maximum number of indexes which will be returned after decoding one frame. CodaDx6 may return 1 for MPEG-4 and H.264.
**normalSliceSize**

Recommended size of buffer used to save slice in normal case. This value is determined by quarter of the memory size for one raw YUV image in KB unit.

**worstSliceSize**

Recommended size of buffer used to save slice in normal case. This value is determined by half of the memory size for one raw YUV image in KB unit.

### 3.2.2.16 DecAvcSliceBufInfo

typedef struct {
    PhysicalAddress sliceSaveBuffer;
    int sliceSaveBufferSize;
} DecAvcSliceBufInfo;

**Description:**

Data structure is used when host application wants to transfer H.264 decoder slice save buffer information.

**sliceSaveBuffer**

Start address of slice save buffer which the decoder can save slice RBSP. This address must be 4 byte-aligned.

Note: This variable is only valid for H.264 decoder case.

**sliceSaveBufferSize**

Size in bytes of a buffer pointed by sliceSaveBuffer

This value must be a multiple of 1024. The maximum size is 65535 x 1024 bytes.

Note: This variable is only valid for H.264 decoder case.
3.2.2.17  DecBufInfo

typedef struct {
    DecAvcSliceBufInfo avcSliceBufInfo;
} DecBufInfo;

Description:
Data structure is used when host application wants to transfer additional buffer information except frame buffer.

*avcSliceBufInfo*

Start address and size of slice save buffer which the decoder can save slice RBSP.

Note: This variable is only valid for H.264 decoder case.

3.2.2.18  DecParam

typedef struct {
    int prescanEnable;
    int prescanMode;
    int dispReorderBuf;
    int iframeSearchEnable;
    int skipframeMode;
    int skipframeNum;
    int chunkSize;
    int picStartByteOffset;
    PhysicalAddress picStreamBufferAddr;
} DecParam;

Description:
Data structure for options of picture decoding,

*prescanEnable*

0 - disable, 1 - enable
If this option is enabled, then decoder performs scanning stream buffers to check whether a full picture stream exists or not. If there is no full picture stream, decoding picture will not be initiated. This option is provided for preventing decoder hanging in advance.

Note: When multiple picture decoding is needed, for example, the first picture decoding with display reordering enabled, pre-scan would not prevent decoder hanging. So in those cases, it would be highly recommended to disable this option.

`prescanMode`

Operation mode of decoder after pre-scan detects a full picture stream

0 – Start decoding, 1 – Returns without decoding

If this option is enabled, decoder returns without picture decoding even though there is a full picture stream in stream buffer. This option is provided for general usage of pre-scan option as a useful tool for stream buffer check.

`dispReorderBuf`

Display frame output enable in display reordering mode

0 – Run decoding, 1 – Display frame output without decoding

If display reordering is enabled, at the end of sequence decoding, several frames not displayed yet will be stored in display frame buffers even though there are no more bitstream data to be decoded. To display all these decoded but not-displayed frames at the end of sequence decoding, application should set this flag by 1. Then decoder will return immediately without decoding, but new frame index for the next display will be given until there no more not-displayed frames in display reordering buffer.

Note: When display reordering is not used, this flag should always be 0.

`iframeSearchEnable`

0 - disable, 1 - enable

If this option is enabled, then decoder performs skipping frame decoding until decoder meet I (IDR) frame. If there is no I frame in given stream, decoder waits for I (IDR) frame. If skipframeNum is n, decoder seeks (n+1)th I (IDR) frame.

Note: When decoder meets EOS (End Of Sequence) code during I-Search, decoder will return -1 (0xFFFF). And if this option is enabled, prescanEnable, prescanMode and skipframeMode options are ignored.

`skipframeMode`

Skip frame function enable and operation mode

0 – Skip frame disable, 1 – Skip frame enabled (skip frames but I (IDR) frame), 2 – Skip frame enabled (skip any frames),
If this option enabled, decoder skip decoding as many as skipframeNum. If skipframNum is 1, prescan function is enabled and prescanMode is 0. If skipframeMode is 1, decoder skip frame but I (IDR) frame and decodes I (IDR) frame. If skipframeMode is 2, decoder skips any frames. After decoder skips frames, decoder return decoded index -2 (0xFFFE) when decoder doesn’t have any frames displayed.

Note: When decoder meets EOS (End Of Sequence) code during frame skip, decoder will return -1 (= 0xFFFF). If this option is enabled, prescanEnable and prescanMode options are ignored.

**skipframeNum**

Number of skip frames

If iframeSearchEnable option enabled, this number means number of skipping I (IDR) frame. If iframeSearchEnable option disabled and skipframeMode option enabled, this number means number of skipping frame.

Note: When this number is 0, skipframeMode option is disabled.

**picStreamBufferAddr**

Physical address of the start address of picture stream buffer in file-play mode

This variable represents the start of a picture stream to be decoded. In file play mode, application might use multiple picture stream buffers for the best performance. By using this variable, application could re-register the start position of the picture stream while issuing a picture decoding operation. This start address of this buffer must be 4-byte aligned, and its size will be specified the following variable, chunkSize.

Note: This variable is only meaningful when both file-play mode and dynamic buffer allocation are enabled.

**picStartByteOffset**

Start byte offset of the picture stream buffer. Since VPU has an internal limitation that stream buffer start address must be 4 byte-aligned, sometimes host application needs to copy the stream data to a separate 4-byte aligned buffer. By using this offset, this kind of overhead could be saved.

Note: This offset should be ranged between 0 and 3.

**chunkSize**

Byte size of a picture stream to be decoded.
This variable represents byte size of a picture stream, and could be read from file container header field. This variable is so crucial in file-play mode operation. In packet-based streaming with ring-buffer, this variable will be ignored.

Note: When this number is 0, skipframeMode option is disabled.

### 3.2.2.19 DecOutputInfo

```c
typedef struct {
    int indexFrameDisplay;
    int indexFrameDecoded;
    int picType;
    int numOfErrMBs;
    PhysicalAddress phys_qpInfo;
    Uint8 *virt_qpInfo;
    int prescanresult;
    int indexFrameNextDecoded[3];
    int notSufficientPsBuffer;
    int notSufficientSliceBuffer;
} DecOutputInfo;
```

**Description:**

Data structure to get information resulting from decoding a frame.

- **indexFrameDisplay**
  
  This is the frame buffer index of a picture to be displayed this time among frame buffers which were registered using VPU_DecRegisterFrameBuffer(). Frame data to be displayed this time will be stored into the frame buffer specified by this index. When delay in display does not exist, this index will always be the same with indexFrameDecoded. But if not, (for example, display reordering in AVC or B-frames in VC-1), this index will not be the same value with indexFrameDecoded.

  If decoder cannot provide an display output at the beginning of sequence decoding with different display order, this index will always have -2 (0xFFFF) or -3 (0xFFFD) depending on decoder skip option. And at the end of sequence decoding, if there is no more output for display, then this value will have -1 (0xFFFF). So by checking this index, host application can easily know whether sequence decoding has been finished or not.

- **indexDecoded**

  The index of the array of frame buffers which were registered using vpu_DecRegisterFrameBuffer(). The frame buffer which this index indicates holds the decoded image.
**picType**

Picture type of the decoded picture, 0 - I picture, 1 - P picture.

**numOfErrMBs**

The number of erroneous macroblocks while decoding a picture.

**qpInfo**

Base address of MB Qstep values with a pre-defined format. The address is the virtual addresss.

**prescanResult**

0 – Incomplete picture stream, 1 – full picture stream exists, 2 – pre-scan disabled

If application enables pre-scan mode for running a picture decoding task, then it should check this flag first of all. If this flag is equal to 0, all the other output information has no meaning. Application should ignore all these output information in this case. Only if prescanResult is bigger than 0, the other output information will be meaningful for application.

**indexFrameNextDecoded[3]**

These are frame buffer indexes which may be used in next VPU_DecStartOneFrame() call. Only indexes as many as nextDecodedIdxNum in DecInitialInfo structure are valid. Application might not call VPU_DecStartOneFrame(), if some of these index is not displayed yet.

**notSufficientPsBuffer**

This is a flag which represents whether PS (SPS/PPS) save buffer is not sufficient to decode current picture. VPU will not get the last part of current picture stream because of buffer overflow. Host must close current instance because the following picture streams cannot be decoded properly because loss of SPS/PPS data.

**notSufficientSliceBuffer**

This is a flag which represents whether slice save buffer is not sufficient to decode current picture. VPU will not get the last part of current picture stream, and will also be macroblock errors because of buffer overflow. Host can continue decoding the remaining pictures of the current input stream without closing current instance, even though several pictures could be error-corrupted.

### 3.2.2.20 SearchRamParam

typedef struct {
   PhysicalAddress searchRamAddr;
}
} SearchRamParam;

**Description:**
This structure is used when host processor set ME search RAM start address.

### 3.2.2.21 VPUMemAlloc

typedef struct {
    ULONG PhysAdd;
    ULONG VirtAdd;
    UINT Reserved;
} VPUMemAlloc;

**Description:**
This structure is used for API function vpu_AllocPhysMem(), PhysAdd is used to store the phy mem address allocate by driver and VirtAdd is used to store the corresponding virtual memory address; Reserved is used by driver internally.

## 3.3 API Definitions

This chapter presents API set for full processing mode of operations by using VPU module.

### 3.3.1 Overview

This section describes the overview of API definitions for VPU module. Basic API architecture is presented first to show the brief overview of VPU API set. And encoder and decoder operation flow based on VPU API functions will be followd.

### 3.3.1.1 Basic Architecture

The VPU API has three basic categories, Control API, Encoder API and Decoder API. Common control API includes API functions for general control of VPU which cannot be categorized into encoder API or decoder API. Initialization function of VPU is a good example of control API.

The VPU API functions are based on a frame-by-frame picture processing scheme. So in order to run a picture encoder or decoder, application should call a proper API function and after completion of processing, application can check the results of picture processing.

In order to support multi-instance encoding and decoding, the API function uses a handle for specify a certain instance and the handle for each instance will be provided when application created a new encoder or decoder instance. If application wants to give a command to a specific instance, the corresponding handle should be used in every API function calls for that instance.
3.3.1.2 Decoder Operation Flow

To decode a bitstream, application must follow the procedure below.

Step 1. Application should open a decoder instance: `vpu_DecOpen()`.

Step 2. Then in order to provide a proper amount of bitstream, first of all application must know the bitstream buffer address: `vpu_DecGetBitstreamBuffer()`.

Step 3. After transferring decoder input stream, application should inform the amount of bits transferred into bitstream buffer: `vpu_DecUpdateBitstreamBuffer()`.

Step 4. Before starting picture decoder operation, applications should know some crucial parameters for decoder operations such as picture size, frame rate, required frame buffer size, etc.: `vpu_DecGetInitialInfo()`.

Step 5. By using the returned frame buffer requirement, application should allocate proper size of frame buffers by calling `vpu_AlocPhymems()`, and inform it to VPU module: `vpu_DecRegisterFrameBuffer()`.


Step 7. Waiting the completion of picture decoder operation by waiting interrupt event.

Step 8. After decoding a frame is complete, application would check the results of decoder operation: `vpu_DecGetOutputInfo()`.

Step 9. If there is more bitstream to decode, go to Step 6, else go to the next step.

Step 10. Free the allocated memory by calling `vpu_FreePhyMems()`, then terminate sequence operation by closing the instance: `vpu_DecClose()`.
Figure 3-1. Decoder Operation Flow
3.3.1.3 Encoder Operation Flow

To encode a bitstream, application must follow the procedure below.

Step 1. Application should open a encoder instance: `vpu_EncOpen`.

Step 2. Before starting picture encoder operation, applications should know some crucial parameters for encoder operations such as required frame buffer size, etc.: `vpu_EncGetInitialInfo`.

Step 3. By using the returned frame buffer requirement, application should allocate proper size of frame buffers by calling `vpu_AllocPhymems`, and inform it to VPU module: `vpu_EncRegisterFrameBuffer`.

Step 4: Generate high-level header syntaxes: `VPU_EncGiveCommand`.

Step 5. Start picture encoder operation picture by picture: `vpu_EncStartOneFrame`.

Step 6. Waiting the completion of picture decoder operation by waiting interrupt event.

Step 7. After encoding a frame is complete, application would check the results of encoder operation: `vpu_EncGetOutputInfo`.

Step 8. If there are more frames to encode, go to Step 4, else go to the next step.

Step 9. Free the allocated memory by calling `vpu_FreePhyMems`, then terminate sequence operation by closing the instance: `vpu_EncClose`.

Note: The Step 4 is necessary for proper operation in decoder side because automatic header insertion is not supported any longer. VOL header in MPEG-4 and SPS/PPS in AVC should be generated once in this step.
Figure 3-2. Encoder Operation Flow
3.3.2 Control API

3.3.2.1 vpu_Init()

Prototype

RetCode vpu_Init(void);

Parameter

None.

Return value

RETCODE_SUCCESS:
Operation was done successfully, which means VPU initialized successfully.

RETCODE_CALLED_BEFORE:
This function call is invalid which means multiple calls of current API function for a given instance are not allowed. In this case, VPU has been already initialized, so that this function call is meaningless and not allowed anymore.

RETCODE_FAILURE:
Operation was failed, which means VPU initialized unsuccessfully.

Description

This function initializes VPU hardware and proper data structures/resources. Application must call this function only once before using VPU module.

3.3.2.2 vpu_Deinit()

Prototype

void vpu_Deinit(void);

Parameter

None.

Return value

None.

Description

This function deinitializes VPU hardware and releases the resources that are allocated in the vpu_Init function. Application must call this function before exit.

3.3.2.3 vpu_IsBusy()

Prototype
int vpu_IsBusy(void);

Parameter

None.

Return value

0: VPU hardware is idle.
Non-zero value: VPU hardware is busy processing a frame.

Description

Application can know whether processing a frame in encoder or decoder is completed or not completed yet.

3.3.2.4 vpu_GetVersionInfo()

Prototype

RetCode vpu_GetVersionInfo(Unit32 *versionInfo);

Parameter

versionInfo: [output] The 16 MSB means product ID and the 16 LSB means firmware version ID.

Return value

RETCODE_SUCCESS:
Operation was done successfully, which means version information is acquired successfully.

RETCODE_FAILURE:
Operation was failed, which means current firmware does not contain any version information.

RETCODE_NOT_INITIALIZED:
VPU was not initialized at all before calling this function. Application should initialize VPU by calling vpu_Init() before calling this function.

RETCODE_FRAME_NOT_COMPLETE:
This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo(). Even though the result of the current frame operation is not necessary, application should call vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo() to proceed this function call.

Description

Application can read out the version information running on the system. This is a new feature supported from firmware version 1.3.0. So in the earlier versions that 1.3.0, this function call will return RETCODE_FAILURE.
3.3.2.5  vpu_AllocPhysMem(

Prototype

RetCode vpu_AllocPhysMem(Uint32 cbSize, VPUMemAlloc *pmemalloc);

Parameter

cbSize:[input] The Number of bytes to allocate

pmemalloc:[output] The Pointer to a VPUMemAlloc that stores the physical/virtual address of the memory allocation.

Return value

If the allocation failed, RETCODE_FAILURE is returned; otherwise, RETCODE_SUCCESS is returned.

Description

This function allocates physically contiguous memory and corresponding virtual memory. For WinCE6.0, when application call this function to allocate memory, driver should allocate contiguous physical memory, reserve a region of pages in the virtual address space for application process.

3.3.2.6  vpu_FreePhysMem()

Prototype

RetCode vpu_FreePhysMem(VPUMemAlloc* pmemalloc);

Parameter

pmemalloc:[input] The Pointer to a VPUMemAlloc that stores the allocated physical/virtual address of the memory.

Return value

If the allocation failed, RETCODE_FAILURE is returned; otherwise, RETCODE_SUCCESS is returned.

Description

This function is called to release physical memory back to the system. For now implementation, this function is just the wrap of FreePhysMem provided by Window CE6.0.

3.3.2.7  vpu_getPhysAddrFromVirtAddr()

Prototype

RetCode vpu_getPhysAddrFromVirtAddr(void* lpvAddress, Uint32 cbSize, PhysicalAddress* lppAddress);

Parameter

lpvAddress:[input] The Pointer to a virtual memory needed to be translated.
cbSize:[input] The Number of bytes to be translated.

lppAddress:[output] The pointer to physical memory corresponding to the lpvAddress.

Return value

If the operation is failed, RETCODE_FAILURE is returned; otherwise, RETCODE_SUCCESS is returned.

Description

This function is used to get physical address from input virtual address. To improve the performance, vpu may need to use display memory directly as frame buffer. We can only get the virtual memory address of display memory, but VPU can only accept contiguous physical memory as the frame buffer. In WinCE6.0, application can use LockPages() to get the physical memory from virtual memory, otherwise in WinCE6.0, LockPages() can only be called in Kernal mode. This API is provided by this use case. In fact it is just a wrapper for LockPages().

3.3.2.8 vpu_ReleaseCodecMutex()

Prototype

BOOL vpu_ReleaseCodecMutex(void);

Parameter

None.

Return value

If the calling failed, then FALSE is returned; otherwise, TRUE is returned.

Description

This function is called to release the ownership of the mutex object created to support multi-instance encoding and decoding. In general decoding case, the calling of vpu_DecStartOneFrame() and vpu_DecGetOutputInfo() must form a pair. After calling the vpu_DecStartOneFrame(), the application must wait for the events with the name of "Vpu PIC Run Command" and "Vpu Bitstream Empty". The vpu_DecGetOutputInfo() must be called when the event with the name of "Vpu PIC Run Command" is signaled. otherwise the event with the name of "Vpu Bitstream Empty" is signaled, and the vpu_ReleaseCodecMutex() must be called instead of vpu_DecGetOutputInfo() to release the ownership of the mutex object. And so the other instances can get the ownership of the mutex to use VPU module.

Now the bit stream buffer empty interrupt isn’t supported in VPU API, so this function will not be used in the application. It only is reserved for future use.

3.3.3 Encoder API

3.3.3.1 vpu_EncOpen()

Prototype
RetCode vpu_EncOpen (EncHandle * pHandle, EncOpenParam * pOpenParam);

Parameter

pHandle: [Output] A pointer to a EncHandle type variable which will specify each instance for application. If no instance is available, null handle will returned.

pOpenParam: [Input] A pointer to a EncOpenParam type structure which describes parameters for the new encoder instance.

Return value

RETCODE_SUCCESS:
Operation was done successfully, which means a new encoder instance was opened successfully.

RETCODE_FAILURE:
Operation was failed, which means getting a new encoder instance was not done successfully. If there is no free instance anymore, this value will be returned in this function call.

RETCODE_INVALID_PARAM:
The given argument parameter, pOpenParam, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

RETCODE_NOT_INITIALIZED:
VPU was not initialized at all before calling this function. Application should initialize VPU by calling vpu_Init() before calling this function.

Description

In order to start a new encoder operation, application must open a new instance for this encoder operation. By calling this function, application can get a handle specifying a new encoder instance. Because the VPU supports multiple instances of codec operations, application needs this kind of handles for the all codec instances now on running. Once application got a handle, application must use this handle to represent the target instances for all subsequent encoder-related functions.

3.3.3.2 vpu_EncClose()

Prototype

RetCode vpu_EncClose(EncHandle pHandle);

Parameter

pHandle: [Input] An encoder handle obtained from vpu_EncOpen().

Return value

RETCODE_SUCCESS:
Operation was done successfully. That means current encoder instance was closed successfully.

RETCODE_INVALID_HANDLE:
This means the given handle for current API function call, \textit{pHandle}, was invalid. This return code might be caused if
\begin{itemize}
  \item \textit{pHandle} is not a handle which has been obtained by \texttt{vpu\_EncOpen()}, for example a decoder handle
  \item \textit{pHandle} is a handle of an instance which has been closed already, etc.
\end{itemize}

**RETCODE\_FRAME\_NOT\_COMPLETE:**

This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling \texttt{vpu\_EncGetOutputInfo()} or \texttt{vpu\_DecGetOutputInfo()}. Even though the result of the current frame operation is not necessary, application should call \texttt{vpu\_EncGetOutputInfo()} or \texttt{vpu\_DecGetOutputInfo()} to proceed this function call.

**Description**

When application finished encoding operations and wanted to release this instance for other processing, application should close this instance by calling this function. After completion of this function call, the instance referred to by \textit{pHandle} will get free. Once application closes an instance, application cannot call any further encoder-specific function with the current handle before re-opening a new instance with the same handle.

### 3.3.3.3 \texttt{vpu\_EncGetBitstreamBuffer()}

**Prototype**

\[
\text{RetCode vpu\_EncGetBitstreamBuffer(EncHandle } \textit{pHandle}, \text{ PhysicalAddress } * \textit{pRdptr}, \\
\text{ PhysicalAddress } * \textit{pWrptr}, \text{ Uint32 } * \textit{size});
\]

**Parameter**

\begin{itemize}
  \item \textit{pHandle}: [Input] A encoder handle obtained from \texttt{vpu\_EncOpen()}.
  \item \textit{pRdptr}: [Output] Stream buffer read pointer for the current encoder instance
  \item \textit{pWrptr}: [Output] Stream buffer write pointer for the current encoder instance
  \item \textit{size}: [Output] A variable specifying the available space in bitstream buffer for the current encoder instance.
\end{itemize}

**Return value**

\begin{itemize}
  \item **RETCODE\_SUCCESS:**
    \begin{itemize}
      \item Operation was done successfully, which means required information for encoder stream buffer was received successfully.
    \end{itemize}
  \item **RETCODE\_INVALID\_HANDLE**
    \begin{itemize}
      \item This means the given handle for current API function call, \textit{pHandle}, was invalid. This return code might be caused if
        \begin{itemize}
          \item \textit{pHandle} is not a handle which has been obtained by \texttt{vpu\_EncOpen()}, for example an decoder handle
        \end{itemize}
    \end{itemize}
\end{itemize}
\( pHandle \) is a handle of an instance which has been closed already, etc.

RETCODE_INVALID_PARAM:

The given argument parameter, \( pRdptr, pWrptr \) or \( size \), was invalid, which means it has a null pointer, or given values for some member variables are improper values.

Description

After encoding frame, application must get bitstream from the encoder. To do that, application must know where bit stream can be get and the maximum size. Application can get the information by calling this function.

3.3.3.4 \textbf{vpu\_EncUpdateBitstreamBuffer()}

Prototype

\[
\text{RetCode vpu\_EncUpdateBitstreamBuffer(EncHandle } pHandle, \text{ Uint32 size);}\]

Parameter

\( pHandle \): [Input] A encoder handle obtained from vpu\_EncOpen().

\( size \): [Input] A variable specifying the amount of bits get from bitstream buffer for the current encoder instance.

Return value

RETCODE_SUCCESS:

Operation was done successfully, which means putting new stream data was done successfully.

RETCODE_INVALID_HANDLE

This means the given handle for current API function call, \( pHandle \), was invalid. This return code might be caused if \( pHandle \) is not a handle which has been obtained by vpu\_EncOpen(), for example an decoder handle \( pHandle \) is a handle of an instance which has been closed already, etc.

RETCODE_INVALID_PARAM:

The given argument parameter, \( size \), was invalid, which means \( size \) is larger than the value obtained from vpu\_EncGetBitstreamBuffer().

Description

Application must let encoder know how much bitstream has been transferred from the address obtained from vpu\_EncGetBitstreamBuffer(). By just giving the size as argument, API automatically handles pointer wrap-around and update read pointer.

3.3.3.5 \textbf{vpu\_EncGetInitialInfo()}

Prototype
RetCode vpu_EncGetInitialInfo(EncHandle pHandle, EncInitialInfo * pInitialInfo);

Parameter

pHandle: [Input] An encoder handle obtained from vpu_EncOpen().

pInitialInfo: [Output] A pointer to a EncInitialInfo type structure which describes required parameters before starting encoder operations.

Return value

RETCODE_SUCCESS:
Operation was done successfully, which means receiving the initial parameters was done successfully.

RETCODE_FAILURE:
Operation was failed, which means there was an error in getting information for configuring the encoder.

RETCODE_INVALID_HANDLE:
This means the given handle for current API function call, pHandle, was invalid. This return code might be caused if

- pHandle is not a handle which has been obtained by vpu_EncOpen(), for example a decoder handle.
- pHandle is a handle of an instance which has been closed already, etc.

RETCODE_FRAME_NOT_COMPLETE:
This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo(). Even though the result of the current frame operation is not necessary, application should call vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo() to proceed this function call.

RETCODE_INVALID_PARAM:
The given argument parameter, pInitialInfo, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

RETCODE_CALLED_BEFORE:
This function call is invalid which means multiple calls of current API function for a given instance are not allowed. In this case, encoder initial information has been received already, so that this function call is meaningless and not allowed anymore.

Description

Before starting encoder operation, application must allocate frame buffers according to the information obtained from this function. This function returns the required parameters for vpu_EncRegisterFrameBuffer(), which will be followed by this function call.
3.3.3.6 vpu_EncRegisterFrameBuffer()

Prototype

RetCode vpu_EncRegisterFrameBuffer (EncHandle pHandle, FrameBuffer * pBuffer,
int num, int stride);

Parameter

pHandle: [Input] An encoder handle obtained from vpu_EncOpen().

pBuffer: [Input] A pointer to a FrameBuffer type structure which describes frame buffer pointer
parameters for the current encoder instance.

num: [Input] A number of frame buffers.

stride: [Input] A stride value of the given frame buffers.

The distance between a pixel in a row and the corresponding pixel in the next row is called “stride.”
The value of stride must be a multiple of 8. The address of the first pixel in the second row does not
necessarily coincide with the value next to the last pixel in the first row. In other words, stride can
have values greater than the picture width in pixels. Application should not set a stride value smaller
than the picture width. So, for Y component, application must allocate at least a space of size (frame
height * stride), and Cb or Cr component, (frame height/2 * stride/2), respectively.

Return value

RETCODE_SUCCESS:

Operation was done successfully, which means registering frame buffers were done successfully.

RETCODE_INVALID_HANDLE:

This means the given handle for current API function call, pHandle, was invalid. This return code
might be caused if
• pHandle is not a handle which has been obtained by vpu_EncOpen(), for example a decoder handle
• pHandle is a handle of an instance which has been closed already, etc.

RETCODE_FRAME_NOT_COMPLETE:

This means frame decoding or encoding operation was not completed yet, so the given API function
call cannot be performed this time. A frame encoding or decoding operation should be completed by
calling vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo(). Even though the result of the current
frame operation is not necessary, application should call vpu_EncGetOutputInfo() or
vpu_DecGetOutputInfo() to proceed this function call.

RETCODE_WRONG_CALL_SEQUENCE:

This means current API function call was invalid considering the allowed sequences between API
functions. In this case, application might call this function before successfully calling
vpu_EncGetInitialInfo(). This function should be called after successfully calling
vpu_EncGetInitialInfo().
RETCODE_INVALID_FRAME_BUFFER:
This means argument pBuffer were invalid, which means it wasn’t initialized yet or not valid anymore.

RETCODE_INSUFFICIENT_FRAME_BUFFERS:
This means the given number of frame buffers, num, was not enough for the encoder operations of the given handle. It should be greater than or equal to the value of minFrameBufferCount obtained from vpu_EncGetInitialInfo().

RETCODE_INVALID_STRIDE:
This means the given argument stride was invalid, which means it is 0, or is not a multiple of 8 in this case.

RETCODE_CALLED_BEFORE:
This function call is invalid which means multiple calls of current API function for a given instance are not allowed. In this case, registering frame buffer for this instance has been done already, so that this function call is meaningless and not allowed anymore.

Description
This function registers frame buffers requested by vpu_EncGetInitialInfo(). The frame buffers pointed to by pBuffer will be managed internally within VPU. These include reference frames, reconstructed frames, etc. Application must not change the contents of the array of frame buffers during the life time of the instance, and num must not be less than minFrameBufferCount obtained by vpu_EncGetInitialInfo().

3.3.3.7 vpu_EncStartOneFrame()

Prototype
RetCode vpu_EncStartOneFrame (EncHandle pHandle, EncParam * pParam);

Parameter
pHandle: [Input] An encoder handle obtained from vpu_EncOpen().

pParam: [Input] A pointer to a EncParam type structure which describes picture encoding parameters for the current encoder instance.

Return value
RETCODE_SUCCESS:
Operation was done successfully, which means encoding a new frame was started successfully.

Note: This return value does not mean that encoding a frame was completed successfully.

RETCODE_INVALID_HANDLE:
This means the given handle for current API function call, pHandle, was invalid. This return code might be caused if
pHandle is not a handle which has been obtained by vpu_EncOpen(), for example a decoder handle

pHandle is a handle of an instance which has been closed already, etc.

RETCODE_FRAME_NOT_COMPLETE:
This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo(). Even though the result of the current frame operation is not necessary, application should call vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo() to proceed this function call.

RETCODE_WRONG_CALL_SEQUENCE:
This means current API function call was invalid considering the allowed sequences between API functions. In this case, application might call this function before successfully calling vpu_EncRegisterFrameBuffer(). This function should be called after successfully calling vpu_EncRegisterFrameBuffer().

RETCODE_INVALID_PARAM:
The given argument parameter, pParam, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

RETCODE_INVALID_FRAME_BUFFER:
This means sourceFrame in input structure EncParam was invalid, which means sourceFrame was not valid even though picture-skip is disabled.

Description
This function starts encoding one frame. Returning from this function does not mean the completion of encoding one frame, but it just initiates encoding one frame. This function should be followed by vpu_EncGetOutputInfo() with the same encoder handle. Before that, application can not call other API functions except for vpu_IsBusy(), vpu_DecGetBitstreamBuffer(), and vpu_DecUpdateBitstreamBuffer().

3.3.3.8 vpu_EncGetOutputInfo()

Prototype
RetCode vpu_EncGetOutputInfo(EncHandle pHandle, EncOutputInfo * pInfo);

Parameter

pHandle: [Input] An encoder handle obtained from vpu_EncOpen().

pInfo: [Output] A pointer to a EncOutputInfo type structure which describes picture encoding results for the current encoder instance.

Return value
RETCODE_SUCCESS:
Operation was done successfully, which means the output information of current frame encoding was received successfully.

RETCODE_INVALID_HANDLE:
The given handle for current API function call, pHandle, was invalid. This return code might be caused if
- pHandle is not a handle which has been obtained by vpu_EncOpen(), for example a decoder handle,
- pHandle is a handle of an instance which has been closed already,
- pHandle is not the same handle as the last vpu_EncStartOneFrame() has, etc.

RETCODE_WRONG_CALL_SEQUENCE:
This means current API function call was invalid considering the allowed sequences between API functions. In this case, application might call this function before successfully calling vpu_EncStartOneFrame(). This function should be called after successfully calling vpu_EncStartOneFrame().

RETCODE_INVALID_PARAM:
The given argument parameter, pInfo, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

Description
This function gets information of the output of encoding. Application can know about picture type, the address and size of the generated bitstream, the number of generated slices, the end addresses of the slices, and macroblock bit position information. Host application should call this function after frame encoding is finished, and before starting the further processing.

3.3.3.9 vpu_EncGiveCommand()

Prototype
RetCode vpu_EncGiveCommand ( EncHandle pHandle, CodecCommand cmd, void *pParam );

Parameter
pHandle: [Input] An encoder handle obtained from vpu_EncOpen().

cmd: [Intput] A variable specifying the given command of CodecCommand type.

pParam: [Input/Output] A pointer to a command-specific data structure which describes picture I/O parameters for the current encoder instance.

Return value
RETCODE_INVALID_COMMAND
This means the given argument, cmd, was invalid which means the given cmd was undefined, or not allowed in the current instance.
**RETCODE_INVALID_HANDLE**

This means the given handle for current API function call, `pHandle`, was invalid. This return code might be caused if

- `pHandle` is not a handle which has been obtained by `vpu_EncOpen()`, for example a decoder handle
- `pHandle` is a handle of an instance which has been closed already, etc.

**RETCODE_FRAME_NOT_COMPLETE**

This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling `vpu_EncGetOutputInfo()` or `vpu_DecGetOutputInfo()`. Even though the result of the current frame operation is not necessary, application should call `vpu_EncGetOutputInfo()` or `vpu_DecGetOutputInfo()` to proceed this function call.

**Description**

This function is provided to let application have a certain level of freedom for re-configuring encoder operation after creating an encoder instance. Some options which can be changed dynamically during encoding a video sequence have been included. Some command-specific return codes are also presented.

**ENABLE_ROTATION**

This command enables rotation of the pre-rotator.

In this case, `pParam` is ignored. This command returns RETCODE_SUCCESS.

**DISABLE_ROTATION**

This command disables rotation of the pre-rotator

In this case, `pParam` is ignored. This command returns RETCODE_SUCCESS.

**ENABLE_MIRRORING**

This command enables mirroring of the pre-rotator

In this case, `pParam` is ignored. This command returns RETCODE_SUCCESS.

**DISABLE_MIRRORING**

This command disables mirroring of the pre-rotator

In this case, `pParam` is ignored. This command returns RETCODE_SUCCESS.

**SET_MIRROR_DIRECTION**

This command sets mirror direction of the pre-rotator, and `pParam` is interpreted as a pointer to `MirrorDirection`. `*pParam` should be one of MIRDIR_NONE, MIRDIR_VER,
MIRDIR_HOR, and MIRDIR_HOR_VER.
MIRDIR_NONE: No mirroring
MIRDIR_VER: Vertical mirroring
MIRDIR_HOR: Horizontal mirroring
MIRDIR_HOR_VER: Both directions
RETCODE_SUCCESS:
Operation was done successfully, which means given mirroring direction is valid.
RETCODE_INVALID_PARAM:
The given argument parameter, \( pParam \), was invalid, which means given mirroring direction is invalid.

SET_ROTATION_ANGLE

This command sets counter-clockwise angle for pre-rotation, and \( pParam \) is interpreted as a pointer to an integer which represents rotation angle in degrees. Rotation angle should be one of 0, 90, 180, and 270.

RETCODE_SUCCESS:
Operation was done successfully, which means given rotation angle is valid.
RETCODE_INVALID_PARAM:
The given argument parameter, \( pParam \), was invalid, which means given rotation angle is invalid.

Note: Rotation angle could not be changed after sequence initialization, because it might cause problems in handling frame buffers.

ENC_GET_SPS_RBSP

This command tells the encoder to generate SPS stream based on the information provided when opening the instance for external usage (for example, “out-of-band” transmission). The generated stream is in RBSP format and in big endian. Argument \( pParam \) is interpreted as a pointer of an EncParamSet type structure. The first variable \( paraSet \) is a physical address where the generated stream is located, and \( size \) is the size of the stream in bytes. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means SPS was successfully generated and it will be available at the received buffer pointer.

RETCODE_INVALID_COMMAND:
This means the given argument, \( cmd \), was invalid which means the given \( cmd \) was undefined, or not allowed in the current instance. In this case, current instance might not be an AVC (H.264) encoder instance.
RETCODE_INVALID_PARAM:
The given argument parameter, \textit{pParam}, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

ENC_GET_PPS_RBSP
This command tells the encoder to generate PPS stream based on the information provided when opening the instance for external usage (for example, “out-of-band” transmission). The generated stream is in RBSP format and in big endian. \textit{pParam} is interpreted as a pointer to an EncParamSet type structure too. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means PPS was successfully generated and it will be available at the received buffer pointer.

RETCODE_INVALID_COMMAND:
This means the given argument, \textit{cmd}, was invalid which means the given \textit{cmd} was undefined, or not allowed in the current instance. In this case, current instance might not be an AVC (H.264) encoder instance.

RETCODE_INVALID_PARAM:
The given argument parameter, \textit{pParam}, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

ENC_PUT_MP4_HEADER
This command tells the encoder to generate headers and insert a certain header at the current bitstream buffer write-pointer. Argument \textit{pParam} is interpreted as a pointer to an EncHeaderParam structure, where \textit{buf} is a physical address pointing the generated stream location, and \textit{size} is the size of generated stream in bytes, and \textit{headerType} is type of header that application want to generate and have values as VOL_HEADER, VOS_HEADER, or VIS_HEADER. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

RETCODE_INVALID_COMMAND:
This means the given argument, \textit{cmd}, was invalid which means the given \textit{cmd} was undefined, or not allowed in the current instance. In this case, current instance might not be an MPEG-4 encoder instance.

RETCODE_INVALID_PARAM:
The given argument parameter, \textit{pParam} or \textit{headerType}, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

ENC_PUT_AVC_HEADER
This command tells the encoder to generate and insert a certain header, SPS or PPS, at the current bitstream buffer write-pointer. The generated stream is byte-stream format NAL unit. Argument \texttt{pParam} is interpreted as a pointer to an EncHeaderParam structure, where \texttt{buf} is a physical address pointing the generated stream location, and \texttt{size} is the size of generated stream in bytes, and \texttt{headerType} is type of header that application want to generate and have values as SPS\_RBSP or PPS\_RBSP. Return values are as follows:

\textbf{RETCODE\_SUCCESS:}

Operation was done successfully, which means the requested header syntax was successfully inserted.

\textbf{RETCODE\_INVALID\_COMMAND}

This means the given argument, \texttt{cmd}, was invalid which means the given \texttt{cmd} was undefined, or not allowed in the current instance. In this case, current instance might not be an AVC (H.264) encoder instance.

\textbf{RETCODE\_INVALID\_PARAM:}

The given argument parameter, \texttt{pParam} or \texttt{headerType}, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

\textbf{ENC\_SET\_SEARCHRAM\_PARAM}

In this case, argument \texttt{pParam} is interpreted as a pointer to a DecParamSet structure. In this case, \texttt{paraSet} is an array of 32 bits which contains PPS RBSP, and \texttt{size} is the size of the stream in bytes. Return values are as follows:

\textbf{RETCODE\_SUCCESS:}

Operation was done successfully, which means transferring a PPS RBSP to decoder was done successfully.

\textbf{RETCODE\_INVALID\_COMMAND:}

The given argument, \texttt{cmd}, was invalid, which means the given \texttt{cmd} was undefined, or not allowed in the current instance. In this case, current instance might not be an AVC (H.264) decoder instance.

\textbf{RETCODE\_INVALID\_PARAM:}

The given argument parameter, \texttt{pParam}, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

\textbf{ENC\_SET\_INTRA\_MB\_REFRESH\_NUMBER}
In this case, argument pParam is interpreted as a pointer to an integer which represents intra refresh number. Intra refresh number should be between 0 and macroblock number of encoded picture. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

ENC_ENABLE_HEC
In this case, argument pParam is ignored. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

RETCODE_INVALID_COMMAND
This means the given argument, cmd, was invalid which means the given cmd was undefined, or not allowed in the current instance. In this case, current instance might not be an MPEG-4 encoder instance.

ENC_DISABLE_HEC
In this case, argument pParam is ignored. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

RETCODE_INVALID_COMMAND
This means the given argument, cmd, was invalid which means the given cmd was undefined, or not allowed in the current instance. In this case, current instance might not be an MPEG-4 encoder instance.
ENC_SET_SLICE_INFO

In this case, argument pParam is interpreted as a pointer to an EncSliceMode structure, where sliceModeuf is a mode which means enabling multi slice structure, and sliceSizeMode is the mode representing mode of calculating one slicesize, and sliceSize is size of one slice. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

RETCODE_INVALID_PARAM:
The given argument parameter, pParam or headerType, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

ENC_SET_GOP_NUMBER

In this case, argument pParam is interpreted as a pointer to an integer which represents GOP number. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

RETCODE_INVALID_PARAM:
The given argument parameter, pParam or headerType, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

ENC_SET_INTRA_QP

In this case, argument pParam is interpreted as a pointer to an integer which represents constant I frame QP. Constant I frame QP should be between 1 and 31 for MPEG-4, between 0 and 51 for AVC (H.264). Return values are as follows:
RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

RETCODE_INVALID_COMMAND
This means the given argument, cmd, was invalid which means the given cmd was undefined, or not allowed in the current instance. In this case, current instance might not be an encoder instance.

RETCODE_INVALID_PARAM:
The given argument parameter, pParam or headerType, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

ENC_SET_BITRATE
In this case, argument pParam is interpreted as a pointer to an integer which represents bitrate. Bitrate should be between 0 and 32767. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

RETCODE_INVALID_COMMAND
This means the given argument, cmd, was invalid which means the given cmd was undefined, or not allowed in the current instance. In this case, current instance might not be an encoder instance.

RETCODE_INVALID_PARAM:
The given argument parameter, pParam or headerType, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

ENC_SET_FRAMERATE
In this case, argument pParam is interpreted as a pointer to an integer which represents frame rate value. Frame rate should be greater than 0. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means the requested header syntax was successfully inserted.

RETCODE_INVALID_COMMAND
This means the given argument, cmd, was invalid which means the given cmd was undefined, or not allowed in the current instance. In this case, current instance might not be an encoder instance.

RETCODE_INVALID_PARAM:
The given argument parameter, pParam or headerType, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

3.3.4 Decoder API

3.3.4.1 vpu_DecOpen()

Prototype
RetCode vpu_DecOpen(DecHandle * pHandle, DecOpenParam * pOpenParam);

Parameter
pHandle: [Output] A pointer to a DecHandle type variable which will specify each instance for application.

pOpenParam: [Input] A pointer to a DecOpenParam type structure which describes required parameters for creating a new decoder instance.

Return value
RETCODE_SUCCESS:
Operation was done successfully, which means a new decoder instance was created successfully.

RETCODE_FAILURE:
Operation was failed, which means getting a new decoder instance was not done successfully. If there is no free instance anymore, this value will be returned in this function call.

RETCODE_INVALID_PARAM:
The given argument parameter, \textit{pOpenParam}, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

\textbf{RETCODE\_NOT\_INITIALIZED:}

This means the VPU was not initialized yet before calling this function. Application should initialize VPU by calling \texttt{vpu\_Init()} before calling this function.

\textbf{Description}

In order to decode, application must “open” the decoder. By calling this function, application can get a handle by which application can refer to a decoder instance. Because VPU is multiple instance codec, application needs this kind of handle. Once application got a handle, application must pass the handle to all subsequent decoder-related functions.

\textbf{3.3.4.2 vpu\_DecClose()}

\textbf{Prototype}

\begin{verbatim}
RetCode vpu\_DecClose(DecHandle pHandle);
\end{verbatim}

\textbf{Parameter}

\begin{itemize}
  \item \textit{pHandle}: [Input] A decoder handle obtained from vpu\_DecOpen().
\end{itemize}

\textbf{Return value}

\begin{itemize}
  \item \textbf{RETCODE\_SUCCESS:}
    \begin{itemize}
      \item Operation was done successfully, which means current decoder instance was closed successfully.
    \end{itemize}
  \item \textbf{RETCODE\_INVALID\_HANDLE}
    \begin{itemize}
      \item This means the given handle for current API function call, \textit{pHandle}, was invalid. This return code might be caused if
      \begin{itemize}
        \item \textit{pHandle} is not a handle which has been obtained by vpu\_DecOpen(), for example an encoder handle
        \item \textit{pHandle} is a handle of an instance which has been closed already, etc.
      \end{itemize}
    \end{itemize}
  \item \textbf{RETCODE\_FRAME\_NOT\_COMPLETE}
    \begin{itemize}
      \item This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling vpu\_EncGetOutputInfo() or vpu\_DecGetOutputInfo(). Even though the result of the current frame operation is not necessary, application should call vpu\_EncGetOutputInfo() or vpu\_DecGetOutputInfo() to proceed this function call.
    \end{itemize}
\end{itemize}

\textbf{Description}

When application finished decoding a sequence and wanted to release this instance for other processing, application should close this instance. After completion of this function call, the instance referred to by \textit{pHandle} will get free. Once application closes an instance, application cannot call any
further decoder-specific function with the current handle before re-opening a new decoder instance with the same handle.

### 3.3.4.3 vpu_DecGetBitstreamBuffer()

**Prototype**

```c
RetCode vpu_DecGetBitstreamBuffer(DecHandle pHandle, PhysicalAddress * pRdptr, PhysicalAddress * pWrptr, Uint32 * size);
```

**Parameter**

- `pHandle`: [Input] A decoder handle obtained from vpu_DecOpen().
- `pRdptr`: [Output] Stream buffer read pointer for the current decoder instance
- `pWrptr`: [Output] Stream buffer write pointer for the current decoder instance
- `size`: [Output] A variable specifying the available space in bitstream buffer for the current decoder instance.

**Return value**

- **RETCODE_SUCCESS**: Operation was done successfully, which means required information for decoder stream buffer was received successfully.
- **RETCODE_INVALID_HANDLE**: This means the given handle for current API function call, `pHandle`, was invalid. This return code might be caused if `pHandle` is not a handle which has been obtained by vpu_DecOpen(), for example an encoder handle
- `pHandle` is a handle of an instance which has been closed already, etc.
- **RETCODE_INVALID_PARAM**: The given argument parameter, `pRdptr`, `pWrptr` or `size`, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

**Description**

Before decoding bit stream, application must feed the decoder with bitstream. To do that, application must know where bit stream can be put and the maximum size. Application can get the information by calling this function. This way is more efficient than providing arbitrary bit stream buffer to the decoder as far as VPU is concerned.

**Note:** The given size is the total sum of free space in ring buffer. So when application downloads bitstream of this given size, Wrptr could meet the end of stream buffer. In this case, application should wrap-around the Wrptr to the beginning of stream buffer, and download the remaining bits. If not, decoder operation could be crashed.
3.3.4.4  vpu_DecUpdateBitstreamBuffer()

Prototype

    RetCode vpu_DecUpdateBitstreamBuffer(DecHandle pHandle, Uint32 size);

Parameter

    pHandle: [Input] A decoder handle obtained from vpu_DecOpen().
    size: [Input] A variable specifying the amount of bits transferred into bitstream buffer for the current decoder instance.

Return value

    RETCODE_SUCCESS:
    Operation was done successfully, which means putting new stream data was done successfully.
    RETCODE_INVALID_HANDLE
    This means the given handle for current API function call, pHandle, was invalid. This return code might be caused if
    • pHandle is not a handle which has been obtained by vpu_DecOpen(), for example an encoder handle
    • pHandle is a handle of an instance which has been closed already, etc.
    RETCODE_INVALID_PARAM:
    The given argument parameter, size, was invalid, which means size is larger than the value obtained from vpu_DecGetBitstreamBuffer(), or than the available space in the bitstream buffer.

Description

    Application must let decoder know how much bitstream has been transferred to the address obtained from vpu_DecGetBitstreamBuffer(). By just giving the size as argument, API automatically handles pointer wrap-around and update write pointer.

3.3.4.5  vpu_DecSetEscSeqInit()

Prototype

    RetCode vpu_DecSetEscSeqInit(DecHandle pHandle, int escape);

Parameter

    pHandle: [Input] A decoder handle obtained from vpu_DecOpen().
    escape: [Input] A flag to enable or disable forced escape from SEQ_INIT.

Return value

    RETCODE_SUCCESS:
Operation was done successfully, which means Force escape flag is successfully provided to BIT Processor.

RETCODE_INVALID_HANDLE

This means the given handle for current API function call, \( pHandle \), was invalid. This return code might be caused if

- \( pHandle \) is not a handle which has been obtained by \( \text{vpu\_DecOpen()} \), for example an encoder handle
- \( pHandle \) is a handle of an instance which has been closed already, etc.

**Description**

This is a special function to provide a way of escaping VPU hanging during DEQ_SEQ_INIT. When this flag is set to 1 and stream buffer empty happens, VPU terminates automatically DEC_SEQ_INIT operation which will be started by calling \( \text{vpu\_DecGetInitialInfo()} \). If target application ensures that high layer header syntax is periodically sent through the channel, application does not need to this option. But if target application cannot ensure that such as file-play, it might be very useful to avoid VPU hanging because of crucial errors in header syntax.

Note: This flag will be applied to all decoder instances together, so it would be highly recommended to reset this flag by 0 after successfully finishing the sequence initialization.

### 3.3.4.6 \texttt{vpu\_DecGetInitialInfo()}

**Prototype**

\[
\text{RetCode vpu\_DecGetInitialInfo(DecHandle pHandle, DecInitialInfo * pInfo);}\]

**Parameter**

- \( pHandle \): [Input] A decoder handle obtained from \( \text{vpu\_DecOpen()} \).
- \( pInfo \): [Output] Pointer to DecInitialInfo data structure.

**Return value**

- **RETCODE_SUCCESS:**
  Operation was done successfully, which means required information of the stream data to be decoded was received successfully.
- **RETCODE_FAILURE:**
  Operation was failed, which means there was an error in getting information for configuring the decoder.
- **RETCODE_INVALID_HANDLE:**
  This means the given handle for current API function call, \( pHandle \), was invalid. This return code might be caused if
• `pHandle` is not a handle which has been obtained by `vpu_DecOpen()`, for example an encoder handle
• `pHandle` is a handle of an instance which has been closed already, etc.

**RETCODE_INVALID_PARAM:**

The given argument parameter, `pInfo`, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

**RETCODE_FRAME_NOT_COMPLETE:**

This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling `vpu_EncGetOutputInfo()` or `vpu_DecGetOutputInfo()`. Even though the result of the current frame operation is not necessary, application should call `vpu_EncGetOutputInfo()` or `vpu_DecGetOutputInfo()` to proceed this function call.

**RETCODE_WRONG_CALL_SEQUENCE:**

This means current API function call was invalid considering the allowed sequences between API functions. In this case, application might call this function before successfully putting bitstream data by calling `vpu_DecUpdateBitstreamBuffer()`. In order to perform this functions call, bitstream data including sequence level header should be transferred into bitstream buffer before calling `vpu_DecGetInitialInfo()`.

**RETCODE_CALLED_BEFORE:**

This function call might be invalid, which means multiple calls of current API function for a given instance are not allowed. In this case, decoder initial information has been already received, so that this function call is meaningless and not allowed anymore.

**Description**

Application must pass the address of a DecInitialInfo structure, where the decoder will store information such as picture size, number of necessary frame buffers, etc. For the details, see definition of DecInitialInfo data structure. This function should be called before starting frame decoding once after creating a decoder instance. It is application’s responsibility to provide sufficient amount of bitstream to the decoder by calling `vpu_DecUpdateBitstreamBuffer()` so that bitstream buffer does not get empty before this function returns.

If application cannot ensure to feed stream enough, application can use Forced Escape option by using `vpu_DecSetEscSeqInit()`.

### 3.3.4.7 `vpu_DecRegisterFrameBuffer()`

**Prototype**

```c
RetCode vpu_DecRegisterFrameBuffer(DecHandle pHandle, FrameBuffer * pBuffer, 
int num, int stride, DecBufInfo * pBufInfo);
```

**Parameter**
pHandle: [Input] A decoder handle obtained from vpu_DecOpen().

pBuffer: [Input] A pointer to a FrameBuffer type structure which describes frame buffer pointer parameters for the current decoder instance.

num: [Input] A number of frame buffers.

stride: [Input] A stride value of the given frame buffers.

pBufInfo: [Input] A pointer to a DecBufInfo type structure which describes information on the additional work buffers. Currently sliceSaveBuffer is only declared by this structure.

Return value

RETCODE_SUCCESS:
Operation was done successfully, which means registering frame buffer information was done successfully.

RETCODE_INVALID_HANDLE:
This means the given handle for current API function call, pHandle, was invalid. This return code might be caused if
- pHandle is not a handle which has been obtained by vpu_DecOpen(), for example an encoder handle
- pHandle is a handle of an instance which has been closed already, etc.

RETCODE_FRAME_NOT_COMPLETE:
This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo(). Even though the result of the current frame operation is not necessary, application should call vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo() to proceed this function call.

RETCODE_WRONG_CALL_SEQUENCE:
This means current API function call was invalid considering the allowed sequences between API functions. In this case, application might call this function before successfully calling vpu_DecGetInitialInfo() before calling this function.

RETCODE_INVALID_FRAME_BUFFER:
This happens when pBuffer was invalid, which means pBuffer wasn’t initialized yet or not valid anymore.

RETCODE_INSUFFICIENT_FRAME_BUFFERS:
This means the given number of frame buffers, num, was not enough for the decoder operations of the given handle. It should be greater than or equal to the value requested by vpu_DecGetInitialInfo().

RETCODE_INVALID_STRIDE:
The given argument \textit{stride} was invalid, which means it is smaller than the decoded picture width, or is not a multiple of 8 in this case.

\textbf{RETCODE_CALLED_BEFORE:}

This function call is invalid which means multiple calls of current API function for a given instance are not allowed. In this case, registering decoder frame buffers has been already done, so that this function call is meaningless and not allowed anymore.

\textbf{Description}

This function is used for registering frame buffers with the acquired information from \texttt{vpu\_DecGetInitialInfo()}. The frame buffers pointed to by \texttt{pBuffer} will be managed internally within VPU. These include reference frames, reconstructed frame, etc. Application must not change the contents of the array of frame buffers during the life time of the instance, and \texttt{num} must not be less than \texttt{minFrameBufferCount} obtained by \texttt{vpu\_DecGetInitialInfo()}.

\subsection{3.3.4.8 \texttt{vpu\_DecStartOneFrame()}}

\textbf{Prototype}

\begin{verbatim}
RetCode vpu_DecStartOneFrame (DecHandle pHandle);
\end{verbatim}

\textbf{Parameter}

\begin{itemize}
  \item \texttt{pHandle}: [Input] A decoder handle obtained from \texttt{vpu\_DecOpen()}.\end{itemize}

\textbf{Return value}

\begin{itemize}
  \item \textbf{RETCODE\_SUCCESS:}
    Operation was done successfully, which means decoding a new frame was started successfully.
    \begin{quote}
    \textbf{Note: This return value does not mean that decoding a frame was completed successfully.}
    \end{quote}
  \item \textbf{RETCODE\_INVALID\_HANDLE:}
    This means the given handle for current API function call, \texttt{pHandle}, was invalid. This return code might be caused if
    \begin{itemize}
      \item \texttt{pHandle} is not a handle which has been obtained by \texttt{vpu\_DecOpen()}, for example an encoder handle
      \item \texttt{pHandle} is a handle of an instance which has been closed already, etc.
    \end{itemize}
  \item \textbf{RETCODE\_FRAME\_NOT\_COMPLETE:}
    This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling \texttt{vpu\_EncGetOutputInfo()} or \texttt{vpu\_DecGetOutputInfo()}. Even though the result of the current frame operation is not necessary, application should call \texttt{vpu\_EncGetOutputInfo()} or \texttt{vpu\_DecGetOutputInfo()} to proceed this function call.
  \item \textbf{RETCODE\_WRONG\_CALL\_SEQUENCE:}
\end{itemize}
This means current API function call was invalid considering the allowed sequences between API functions. In this case, application might call this function before successfully calling vpu_DecRegisterFrameBuffer(). This function should be called after successfully calling vpu_DecRegisterFrameBuffer().

RETCODE_DEBLOCKING_OUTPUT_NOT_SET:

This return code can happen, when deblinking filter option is activated but required deblinking output information is not available. If deblinking filter is enabled in MPEG-4 case, application should register the frame buffer information of deblinking filtered output by using vpu_DecGiveCommand().

Description

This function starts decoding one frame. Returning from this function does not mean the completion of decoding one frame, but it just initiates decoding one frame. Every call of this function should be matched with vpu_DecGetOutputInfo() with the same handle. Before that, application can not call other API functions except for vpu_IsBusy(), vpu_DecGetBitstreamBuffer(), and vpu_DecUpdateBitstreamBuffer().

When application uses pre-scan mode, possibility of happening decoder hanging will approaches to zero.

3.3.4.9 vpu_DecGetOutputInfo()

Prototype

RetCode vpu_DecGetOutputInfo(DecHandle pHandle, DecOutputInfo * pInfo);

Parameter

pHandle: [Input] A decoder handle obtained from vpu_DecOpen().

pInfo: [Output] A pointer to a DecOutputInfo type structure which describes picture encoding results for the current decoder instance.

Return value

RETCODE_SUCCESS:

Operation was done successfully, which means receiving the output information of current frame was done successfully.

RETCODE_INVALID_HANDLE:

This means argument pHandle is invalid. This includes cases where pHandle is not a handle which has been obtained by vpu_DecOpen(), pHandle is a handle to an instance already closed, or pHandle is a handle to an decoder instance. Also,this value is returned when vpu_DecStartOneFrame() is matched with vpu_DecGetOutputInfo() with different handles.

RETCODE_WRONG_CALL_SEQUENCE:
This means current API function call was invalid considering the allowed sequences between API functions. In this case, vpu_DecStartOneFrame() with the same handle might not be called before calling this function.

RETCODE_INVALID_PARAM:
The given argument parameter, pInfo, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

Description
Application can get the information of output of decoding. The information includes the information of a frame buffer which contains a reconstructed image. Host application should call this function after frame decoding is finished, and before starting the further processing.

Note: If pre-scan mode is enabled, first of all, application should check prescanResult. If the value of prescanResult \(= 0\), all the other output information has no meaning at all.

3.3.4.10 vpu_DecBitBufferFlush()

Prototype
RetCode vpu_DecBitBufferFlush(DecHandle pHandle);

Parameter
\(pHandle\): [Input] A decoder handle obtained from vpu_DecOpen().

Return value
RETCODE_SUCCESS:
Operation was done successfully, which means receiving the output information of current frame was done successfully.

RETCODE_FRAME_NOT_COMPLETE:
This means frame decoding or encoding operation was not completed yet, so the given API function call cannot be performed this time. A frame encoding or decoding operation should be completed by calling vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo(). Even though the result of the current frame operation is not necessary, application should call vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo() to proceed this function call.

RETCODE_INVALID_HANDLE:
This means argument \(pHandle\) is invalid. This includes cases where \(pHandle\) is not a handle which has been obtained by vpu_DecOpen(), \(pHandle\) is a handle to an instance already closed, or \(pHandle\) is a handle to an decoder instance. Also, this value is returned when vpu_DecStartOneFrame() is matched with vpu_DecGetOutputInfo() with different handles.

RETCODE_WRONG_CALL_SEQUENCE:
This means current API function call was invalid considering the allowed sequences between API
functions. In this case, vpu_DecRegisterFrameBuffer() with the same handle might not be called
before calling this function.

Description
Application can flush bitstream which exist in decoder bitstream buffer without decoding of each
instance. If bitstream buffer flushed, read pointer and write pointer of bitstream buffer of each instance
are set bitstream buffer start address.

3.3.4.11 vpu_DecGiveCommand()

Prototype
RetCode vpu_DecGiveCommand(DecHandle pHandle, CodecCommand cmd, void * pParam);

Parameter
pHandle: [Input] A decoder handle obtained from vpu_DecOpen().

cmd: [Input] A variable specifying the given command of CodecComand type.

pParam: [Input/Output] A pointer to a command-specific data structure which describes picture I/O
parameters for the current decoder instance.

Return value
RETCODE_INVALID_COMMAND
The given argument, cmd, was invalid, which means the given cmd was undefined, or not allowed in
the current instance.

RETCODE_INVALID_HANDLE
This means the given handle for current API function call, pHandle, was invalid. This return code
might be caused if
• pHandle is not a handle which has been obtained by vpu_DecOpen(), for example an encoder
  handle

pHandle is a handle of an instance which has been closed already, etc.

RETCODE_FRAME_NOT_COMPLETE
This means frame decoding or encoding operation was not completed yet, so the given API function
call cannot be performed this time. A frame encoding or decoding operation should be completed by
calling vpu_EncGetOutputInfo() or vpu_DecGetOutputInfo(). Even though the result of the current
frame operation is not necessary, application should call vpu_EncGetOutputInfo() or
vpu_DecGetOutputInfo() to proceed this function call.

Description
This function is provided to let application have a certain level of freedom for re-configuring decoder
operation after creating an decoder instance. Some options which can be changed dynamically during
decoding a video sequence have been included. Some command-specific return codes are also presented.

**ENABLE_ROTATION**

This command enables rotation of the post-rotator. In this case, \(p\text{Param}\) is ignored. This command returns RETCODE_SUCCESS.

**DISABLE_ROTATION**

This command disables rotation of the post-rotator. In this case, \(p\text{Param}\) is ignored. This command returns RETCODE_SUCCESS.

**ENABLE_MIRRORING**

This command enables mirroring of the post-rotator. In this case, \(p\text{Param}\) is ignored. This command returns RETCODE_SUCCESS.

**DISABLE_MIRRORING**

This command disables mirroring of the post-rotator. In this case, \(p\text{Param}\) is ignored. This command returns RETCODE_SUCCESS.

**SET_MIRROR_DIRECTION**

This command sets mirror direction of the post-rotator, and \(p\text{Param}\) is interpreted as a pointer to MirrorDirection. *\(p\text{Param}\) should be one of MIRDIR_NONE, MIRDIR_VER, MIRDIR_HOR, and MIRDIR_HOR_VER.

- MIRDIR_NONE: No mirroring
- MIRDIR_VER: Vertical mirroring
- MIRDIR_HOR: Horizontal mirroring
- MIRDIR_HOR_VER: Both directions

RETCODE_SUCCESS:
Operation was done successfully, which means given mirroring direction is valid.

RETCODE_INVALID_PARAM:
The given argument parameter, \(p\text{Param}\), was invalid, which means given mirroring direction is invalid.

**SET_ROTATION_ANGLE**

This command sets counter-clockwise angle for post-rotation, and \(p\text{Param}\) is interpreted as a pointer to an integer which represents rotation angle in degrees. Rotation angle should be one of 0, 90, 180, and 270.

RETCODE_SUCCESS:
Operation was done successfully, which means given rotation angle is valid.
RETCODE_INVALID_PARAM:
The given argument parameter, `pParam`, was invalid, which means given rotation angle is invalid.

SET_SPS_RBSP
This command applies SPS stream to the decoder received from a certain “out-of-band” reception scheme. The stream should be in RBSP format and in big Endian. Argument `pParam` is interpreted as a pointer to a DecParamSet structure. In this case, `paraSet` is an array of 32 bits which contains SPS RBSP, and size is the size of the stream in bytes. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means transferring a SPS RBSP to decoder was done successfully.

RETCODE_INVALID_COMMAND:
The given argument, `cmd`, was invalid, which means the given `cmd` was undefined, or not allowed in the current instance. In this case, current instance might not be an AVC (H.264) decoder instance.

RETCODE_INVALID_PARAM:
The given argument parameter, `pParam`, was invalid, which means it has a null pointer, or given values for some member variables are improper values.

SET_PPS_RBSP
This command applies PPS stream to the decoder received from a certain “out-of-band” reception scheme. The stream should be in RBSP format and in big Endian. Argument `pParam` is interpreted as a pointer to a DecParamSet structure. In this case, `paraSet` is an array of 32 bits which contains PPS RBSP, and size is the size of the stream in bytes. Return values are as follows:

RETCODE_SUCCESS:
Operation was done successfully, which means transferring a PPS RBSP to decoder was done successfully.

RETCODE_INVALID_COMMAND:
The given argument, `cmd`, was invalid, which means the given `cmd` was undefined, or not allowed in the current instance. In this case, current instance might not be an AVC (H.264) decoder instance.

RETCODE_INVALID_PARAM:
The given argument parameter, `pParam`, was invalid, which means it has a null pointer, or given values for some member variables are improper values.
Chapter 4 How to Control VPU

This section presents the detailed information about VPU control scheme based on API functions with some practical programming issues.

- VPU Initialization
- Encoder Control
- Decoder Control

4.1 VPU Initialization

When host processor turns on VPU first time, host processor should do the following operations for future use of VPU, so called initialization process. All these operations could be done easily by calling a single API function, `vpu_Init()`.

- Disable BIT Processor by setting `BIT_CODE_RUN (BASE + 0x000) = 0`,
• Write BIT Processor micro-code to SDRAM which is accessible by VPU directly during run-time,
• Download first N KB micro-code to BIT Processor code memory directly,
• Set BIT Processor buffer pointers, Working Buffer, Parameter Buffer & Code Buffer,
• Set Stream Buffer Control Options & Frame Buffer Endian Mode,
• Interrupt Enable & Reset Registers,
• Enable BIT Processor by setting, BIT_CODE_RUN register = 1

Detailed information about each initialization steps and some programming tips will be presented in the following sub-sections.

4.1.1 Version Check of BIT Processor Microcode

Application can check the version information of BIT processor microcode during runtime. The version number of microcode is presented by a 32 bit value. The 16 MSB means the product number defined by IP internally, and the 16 LSB means the version number specified by the following rule:

\[
\begin{align*}
\text{Version_number}[15:12] & : \text{Major revision}, \\
\text{Version_number}[11:8] & : \text{Minor revision}, \\
\text{Version_number}[7:0] & : \text{Revision Patches}.
\end{align*}
\]

So this version number can have a value from 0.0.0 to 15.15.255. A dedicated command will be used for this version check, and this will be supported by using vpu_GetVersionInfo() function after initialization.

4.1.2 BIT Processor Enable and Disable

BIT Processor has a dedicated register which activates or de-activates BIT Processor during run-time, BIT_CODE_RUN (BASE + 0x000). During initialization, BIT processor program memory will be updated and some configuration registers for controlling VPU operation will also be set. So during this process, BIT processor should be disabled just for safety.

After finishing all initialization process, host processor will enable BIT processor. Then BIT processor internally starts its own initialization process and will be ready for working.

4.1.3 BIT Processor Data Buffer Management

BIT Processor requires certain amount of SDRAM space for its codec operations. This dedicated memory space for BIT processor will include memory space for BIT processor microcode, internal work buffer, some parameter buffers, etc. The size of each sub-buffer should be given as follows:

\[
\begin{align*}
\text{#define FMO_SLICE_SAVE_BUF_SIZE}(32) \\
& \text{// internal buffer for FMO slice group data store in H.264 encoder} \\
& \text{// if FMO is not used in encoder, this buffer size could be set by 0} \\
\text{#define CODE_BUF_SIZE}(64*1024) & \text{// byte size of Code buffer} \\
\text{#define WORK_BUF_SIZE} & \text{((288 * 1024)+(FMO_SLICE_SAVE_BUF_SIZE * 1024 * 8))}
\end{align*}
\]
#define PARA_BUF_SIZE (10*1024) // byte size of Parameter Buffer
#define PARA_BUF2_SIZE(1728)

And as well, there is another buffer which should be reserved by host processor, which will be used by API layer for providing output parameters according to application specific parameters. Its size is 1728 byte and it is automatically placed between Work Buffer and Parameter Buffer in order to reserve a safe guard region for Work Buffer. So total size of dedicated data buffer for BIT processor will be,

\[(64 + 288 + 8*32+10) * 1024 + 1728 = 634,560 \text{ Byte}.\]

In VPU API, the initialization function only receives the start address of this internal buffer as argument. That means, 634,560 byte space starting from the start address should be the dedicated memory space for VPU, and we recommend that any other process should not access this memory space while VPU is enabled. So it is highly recommended for host processor to reserve the specified size of dedicated buffer for BIT processor, and call `vpu_Init()` with the start address of the reserved memory.

In addition to above sub-buffers, VPU needs buffers for saving SPS/PPS and SLICE RBSP when decodes H.264 stream. In general, 5KB is sufficient as SPS/PPS save buffer size and quarter of raw YUV image size is sufficient as SLICE save buffer. If VPU needs more buffer to decode H.264 stream, VPU will report buffer overflow.

All the start address of internal buffer partitions, Code Buffer, Work Buffer and Parameter Buffer, will be calculated inside of `vpu_Init()` function. And those calculated start address will be set on the dedicated registers defined on Host Interface Register automatically in `vpu_Init()`.

### 4.1.4 BIT Processor Microcode Management

BIT Processor has its own program memory inside of VPU, but the content of this program memory will be updated according to the required codec standard dynamically. The advantage of this dynamic microcode reloading will be the reduction of program memory size. This advantage is so meaningful because BIT processor generally requires many sets of microcode in order to support several codec stands in duplex mode. Generally speaking, it seldom happens that codec standard is changed in the middle of codec application. So dynamic reloading for changing codec is not a burden in cycle consumption at all. In worst case, dynamic code reloading can happen once per picture processing, but considering the amount of maximum reloaded code, it will not be a big burden either to VPU cycle consumption.

Since dynamic reloading is done by VPU itself, the only thing host processor should do is copying the given microcode to the reserved Code Buffer before initializing VPU. Of course, the first loading of microcode to BIT Processor’s program memory should be done by host processor separately.

### 4.1.5 Stream Buffer Management

Stream buffer is a shared buffer between host processor and VPU both in encoder case and decoder case. C&M video codec IP normally uses ring-buffers for both encoder and decoder to maximize the efficiency of stream buffer of fixed size.

In encoder case, VPU writes stream data on stream buffer and host process just reads the output bitstream. On the contrary, in decoder case, VPU reads stream data from stream buffer, while host processor puts
stream data to stream buffer. This one directional data flow enables to control stream buffer by using a read pointer and a write pointer only. By calculating the displacement between these two pointers, host can get very easily available stream buffer space in decoder case or available new stream data in encoder case.

However, in C&M video IP, there is an internal constraint that minimum data transfer unit between stream buffer and internal bitstream handling block, GBU, is 512 byte. So if the stream data size is not a multiple of 512 byte, there will be a special way to handling it.

In encoder case, there is no way for host processor to flush out all the generated bitstream data to stream buffer without any special option. So the last part of encoded picture stream might be moved into stream buffer only after the next picture encoding operation is started. In order to avoid this undesirable situation for application developers who want to get encoded stream frame by frame, C&M video encoder HW provides a very special way to flush out all the generated bitstream at the end of picture encoding.

When encoder reserves a limited size of encoder stream buffer, a ring-buffer scheme will be needed. By using two dedicated functions, \texttt{vpu\_EncGetBitStreamBuffer()} and \texttt{vpu\_EncUpdateBitStreamBuffer()}, application can easily handle the read pointer in encoder case while accessing encoder bitstream buffer. And moreover, host processor can choose an additional option of resetting buffer write pointer to the start address of bitstream buffer (\texttt{streamBufResetEnable} member of \texttt{EncOpenParam} is enabled). When this option is enabled, stream buffer is not working as a ring-buffer.

In decoder case, C&M video IP does provide similar options now to support frame-based picture decoding option such as file play mode with container structure and chunk size. In order to use this file play mode, decoder stream buffer should be big enough to store more than one picture. It could be a big burden in some streaming based application, but in local player applications, it would be desirable to use this mode. So currently, we provide two streaming mode in decoder operation, packet-based streaming and frame-based streaming.

In packet-based streaming, decoder stream buffer is managed as a ring-buffer whose read pointer and write pointer is automatically wrapping around at the boundaries. When application is going to download new chunk of bitstream, application should check the available space in bitstream buffer. Even though the available space could be easily computed from read pointer, write pointer and buffer size, VPU API provides a dedicated function for informing buffer read pointer, buffer write pointer and the available space in stream buffer, \texttt{vpu\_DecGetBitStreamBuffer()}. Based on the returned value from this API function, application could download a new chunk of bitstream whose size should be smaller than the available buffer space.

In frame-based streaming, decoder stream buffer is generally managed as a line-buffer because it is safer and more convenient in application side. \texttt{So in frame-based streaming mode, only line-buffer scheme is allowed in CodaDx6 VPU}. Application always knows the size of an encoded frame (chunk size) from file format header syntax. So application can copy one chunk from the start address of the decoder stream buffer and start picture decoding operation.

Note: The write pointer should always be reset by the start address of the stream buffer by application before starting decoder operation in frame-based streaming mode.

As well as these options, host processor can choose the Endian option of stream buffer, and can enable or disable buffer full/empty check option. All these options for stream buffer data management will be stored
into a dedicated Host Interface Register, \texttt{BIT\_BITSTREAM\_CTRL}, and will be referred by BIT Processor during run-time.

### 4.1.6 Interrupt Signaling Management

In order to achieve maximum efficiency in VPU control, VPU IP basically provides interrupt signaling for completion of a requested operation as well as stream buffer empty/full. But for some commands returning quickly, interrupt signaling is not so helpful in that case.

Currently, Video Codec IP provides interrupt signaling for the following commands:

- \texttt{BIT\_RUN\_COMPLETE}
  
  BIT Processor Initialization complete after setting \texttt{BIT\_CODE\_RUN} by 1,

- \texttt{ENC/DEC\_SEQ\_INIT}
  
  Encoder or Decoder Sequence Initialization complete,

- \texttt{ENC/DEC\_SEQ\_END}
  
  Encoder of Decoder Sequence Termination complete,

- \texttt{ENC/DEC\_PIC\_RUN}
  
  Encoder or Decoder Picture Processing complete,

- \texttt{ENC/DEC\_SET\_FRAME\_BUF}
  
  Encoder or Decoder Frame Buffer registration complete,

- \texttt{ENC\_HEADER}
  
  Encoder insertion of header syntax complete,

- \texttt{ENC\_PARA\_SET}
  
  Encoder SPS/PPS acquisition complete

- \texttt{DEC\_PARA\_SET}
  
  Decoder SPS/PPS transfer complete

- \texttt{DEC\_BUF\_FLUSH}
  
  Flushing decoder stream buffer complete

Among them, \texttt{DEC\_SEQ\_INIT} and \texttt{DEC\_PIC\_RUN} can cause VPU stuck when input bitstream is not enough. So if application enables bitstream buffer-empty interrupt with these interrupts, it will be very helpful to avoid unnecessary cycle consumptions in host application.

Each interrupt could be easily enabled or disabled by writing 0 or 1 to the corresponding bit field of Interrupt Enable Register. And when an interrupt is signaled, application can check the source of interrupt by checking the value of Interrupt Reason Register.

All these kinds of interrupt signaling could be replaced by a polling scheme of reading BIT processor Busy-Flag, when interrupt signaling is not easily applicable.
In current API implementation, only the ENC/DEC_PIC_RUN interrupt is concerned by the Application. Other interrupt are used by API internal or not supported.

4.2 Encoder Control

4.2.1 Creating an Encoder Instance

After initialization of VPU, the first step to run encoder operation should be creation of an encoder instance, and acquisition of a handle for specifying that encoder instance. It could be easily done by using a single API function called vpu_EncOpen().

When creating a new encoder instance, application should specify the internal features of this encoder instance through EncOpenParam structure. This structure includes the following information about the new encoder instance:

- Bitstream buffer address & size
  Physical address of bitstream buffer start address and its size.
- Codec standard
  Video codec standard such as H.263, MPEG-4 or H.264.
- Picture size
  Picture width and height.
- Target frame rate and target bitrate with VBV model parameters, initialDelay and vbvBufferSize
  Video Buffer Verifier (VBV) mode parameters are optional even though rate control is enabled.
- Gop size
  Frequency of periodic intra (or IDR) pictures in encoded stream output.
- Slice enable/disable, slice size mode and slice size
  Slice mode enable or disable as well as slice size with slice size mde (number of bits or number of MB’s in each slice).
- Output report such as sliceReport, mbReport, etc. (Currently, sliceReport and mbReport options are only supported.)

Informative output data such as slice boundary, MB boundary in encoded bitstream. More encoded output data will be added on according to the requirements of target application.

- Misc. options such as enableAutoSkip and intraRefresh
  Enable auto-skipping of pictures when output bit count is big enough as well as enable intra-refresh for error robustness and the number of intra MB’s in a non-intra picture.
- Ring buffer mode enable
  Application can decide whether a ring-buffer based streaming scheme is used or not. When this option is disabled, then a frame-based streaming scheme will be used with a line-buffer scheme.
- Dynamic buffer allocation enable
Application can allocate picture stream buffer dynamically by enabling dynamic buffer allocation only if ring-buffer mode is disabled. If dynamic buffer allocation is disabled, the address and size of bitstream buffer specified given with this API will be used in picture encoding. If dynamic buffer allocation is enabled, the address and the size of picture stream buffer can be dynamically given by application while issuing picture encoding operation.

- Intra Quantization Step

Intra Qstep value can be configurable by specifying this value greater than 0. Even if rate control is enabled, VPU encoder will use this fixed quantization step for all I-frames. This intra quantization step could be re-configurable after creating an instance dynamically.

- Video standard specific parameters

Specify standard-specific parameters for each video codec standard such as error resilience tools in MPEG-4, Annexes in H.263, deblocking parameters in H.264, etc.

By using these options, application can get a well optimized output for the requirements of target application. Especially, some output information options such as sliceReport, mbReport, etc. can give a very useful tool for application developer which struggles with lots of constraints in target applications. For example, in case that packet size is fixed, application might need to insert one slice to a certain amount of bits. If slice size is given by the number of bits, it does not ensure that the output slice size is smaller than the given size because of the variable length characteristics of encoding process. So application should divide a slice into two packets frequently, then it might cause big inefficiency in packetization. In order to achieve an easy packetization, application can set slice size by (packet_size – N) with a certain margin of N, which allows the output slice size to be less than the packet size. Then application can easily put a slice into a packet by just referring the slice boundary information provided by VPU as encoder output.

Flexible Macroblock Ordering (FMO) is a special error resilience tool for H.264. TRISTANeX VPU supports two most popular FMO types, interleaved and dispersed. By specifying three variables, fmoEnable, fmoType, and fmoSliceNum, in EncOpenParam structure, application can easily enable FMO options in encoder operation. The variable, fmoType, should be 0 or 1, and the number of slice groups in FMO, fmoSliceNum, should be between 2 and 8.

After creating an encoder instance with these parameters, application cannot change any initial parameters specified in this stage. Whenever application wants to change any of these basic parameters, application should close this instance and re-create another encoder instance with new initial parameters.

The API function, vpu_EncOpen(), does not require any operations on VPU side but declares all the internal parameters to be used in later stage as well as bitstream buffer information.

### 4.2.2 Configuring VPU for Encoder Instance

#### 4.2.2.1 Sequence Initialization

After registering all the required information for the new encoder instance, host application should configure VPU to be ready for supporting the new encoder instance. This procedure will be done by
setting all the encoder related information to VPU Host Interface Registers and give a command, ENC_SEQ_INIT, to VPU for initiating internal configuration operation in VPU. All this process is mainly done by an API function, vpu_EncGetInitialInfo() and this function return a very crucial output parameter for encoder operations, “the minimum number of frame buffers.” Normally, function return a very crucial output parameter for encoder operations, “the minimum number of frame buffers.” Normally, this process does not require so much time, and it should be done only once at the beginning of encoder instance. So it is not recommended to use interrupt signaling for this function, but interrupt signaling is allowed after completion of this operation by enabling the corresponding bit on Interrupt Enable Register.

4.2.2.2 search ram configuration

As well as this mandatory process for configuring VPU, some more operations could be required depending on the VPU optional configuration. For example, some video codec IP requires the start address of Search RAM which will be used in Motion Estimation block in VPU. By default, it will be set by 0 and it would not cause any problems in host applications, which do not share SRAM with VPU. But depending on application requirements, the Search RAM area should be configurable for sharing SRAM memory between host application and VPU ME block. There are two dedicated registers for storing Search RAM base address and Search RAM size. The Search RAM size will be determined by the width of encoded picture and it would be different from the video codec IP HW architecture.

Instead of writing those registers directly, by using vpu_EncGiveCommand() function with SET_ENC_SEARCHRAM_PARAM command and Search RAM base address, application can configure Search RAM parameters very easily.

4.2.2.3 Registering Frame Buffers

This configuring process will be finished by registering frame buffers to VPU for its future picture encoding operation. In this final stage of configuration, the returned parameter from vpu_EncGetInitialInfo(), “the minimum number of frame buffer”, will have a very important meaning. This parameter means that application should reserve at least the same number of frame buffers to VPU for proper encoding operation.

4.2.2.4 Generating High-level header syntaxes

When opening an encoder instance is complete by calling VPU_EncGetInitialInfo(), application MUST generate the high-level header syntaxes such as VOS/VO/VOL headers in MPEG-4 and SPS/PPS in AVC from VPU by using VPU_EncGiveCommand(). These high-level syntaxes can also be used directly for negotiation in transport protocol layer of application.

There are two possible ways for generating these header syntaxes via PARA_BUF or via stream buffer. The recommended way for getting header syntaxes is to use ENC_PUT_AVC/MP4_HEADER command via stream buffer. If application uses this set of commands, then the resulting header syntaxes will be stored into bitstream buffer according to the given Endian setting.
If DecBufReset is enabled, the output header syntaxes will be written onto bitstream buffer starting from the base address of bitstream buffer. If application does not read out each header-syntax one by one, they will be overwritten by the following header syntaxes. If application wants to read out a set of header syntaxes (such as VOS/VO/VOL or SPS/PPS), then application should disable DecBufReset and enable DecBufFlush bit. After completing the generation of the last header syntax, then application can read out a cascaded set of header syntaxes together.

The other way for generating header syntaxes via PARA_BUF will be helpful when application just want to generate header syntaxes in the middle of encoding. It can be done by using ENC_GET XXX HEADER in MPEG-4 case, and ENC_GET XXX RBSP in AVC case. Regardless of streaming mode, this command will generate header syntaxes successfully. But in this case, Endian setting will always be Big-Endian. So on Little-Endian system, Endian conversion should be performed.

4.2.3 Running Picture Encoder On VPU

4.2.3.1 YUV Input Loading

Before running a picture encoder operation, host application should provide a 4:2:0 formatted input YUV image with pre-defined size. The new input image might be coming from external video input device like CMOS sensor. In order to preventing VPU from idling while waiting for completion of receiving input picture, it is recommended to use dual buffering scheme of input image. Then encoder will not spend any cycles for idling before starting its operation.

4.2.3.2 Initiating Picture Encoding

When activating picture encoding operation, application should provide the following information to VPU:

- Source frame address
- Base address of each component of input YUV picture
- Quantization step
- Quantization step for the current picture which will be ignored when rate control is enabled
- Forced Frame Skip and Forced I-picture options
  - Forced frame skip means skipping current frame encoding unconditionally and force I-picture means encoding current frame as I-frame unconditionally

After setting all these information to VPU, host process can initiate picture encoding operation by sending ENC_PIC_RUN command to VPU. All of these pre-defined processor can be performed by just calling a single API function, vpu_EncStartOneFrame() with EncParam structure. This API
function just initiates picture encoding operation, and **returning from this API does not mean that picture encoding is completed.**

Quantization step size given to VPU with ENC_PIC_RUN will only be meaningful when rate control option during opening encoder instance is disabled. This additional feature is provided to support application-specific VBR encoder operations. Forced frame skip option is so useful when external situation of encoder and encoding a new picture is not allowed temporarily. Automatic frame skipping in VPU rate control is used for limiting the output amount of bitstream under the given target bit-rate. On the contrary, Forced frame skip will be used by application when encoding a picture is problematic under a certain external situation. For example, let’s assume that channel condition is temporarily too bad and transmitting encoded stream is impossible. Then application can suspend encoder operation for a while by using this Forced frame skip option.

Forced I-frame option is another useful option of application when the remote receiver side reports an error during decoder operation. Even though a certain error concealment or error robustness scheme might be implemented on decoder side, the best way to recover in decoder error is to send an I-frame. So by using this Forced I-frame option, application can achieve error-recovery of the remote receiver side very effectively.

### 4.2.3.3 Completion of Picture Encoding

Picture encoder operation will took a certain amount of time, and application could go on other tasks while waiting for the completion of picture encoding operation such as packetization of encoded stream for transmission. Application can use two different type of scheme for detecting completion of picture encoding operation, polling a status register or interrupt signaling. When application is going to use a polling scheme, application just needs to check the BusyFlag Register of BIT processor. Calling **vpu_IsBusy()** will give the same result.

Interrupt signaling could be the most efficient way to check the completion of a given command. An interrupt signal for ENC_PIC_RUN command is mapped on bit[3] of Interrupt Enable Register. So application could easily know the completion of picture encoder operation from this dedicated interrupt signal from VPU.

### 4.2.3.4 Encoder Stream Handling

When encoder stream buffer is big enough to store any big size of picture stream, then encoder does not need to get any bitstream during picture encoder operation. After encoder operation is finished, host application can read the encoded bitstream according to the requirements of packetization.

But when encoder stream buffer is not big enough to store a whole picture stream, then encoder buffer-full could happen and until this buffer-full situation is resolved, encoder task running on VPU will be hanged. So while picture encoding is running on, application should keep on reading out encoded bit stream from stream buffer in order to avoid this undesirable encoder hanging.

When using a ring-buffer scheme with a limited size of encoder stream buffer, stream reading during encoder operation is recommended. By using two dedicated functions, **vpu_EncGetBitStreamBuffer()** and **vpu_EncUpdateBitStreamBuffer()**, application can easily handle the read pointer in encoder case while accessing encoder bitstream buffer. But if
ENC_BUFFER_RESET option is used with a big stream buffer enough to store one encoded picture data, then host could read the encoded bitstream at the end of each picture encoding only. In this case, application could do safely other tasks while picture encoding is running on VPU. vpu_EncGetBitStreamBuffer() and vpu_EncUpdateBitStreamBuffer() have no meaning when application uses ENC_BUFFER_RESET option.

4.2.3.5 Acquiring Encoder Results

Whenever a picture encoding is completed, host application can get the encoded output such as encoded picture type, number of slices, etc. And according to the input parameter setting of picture encoding, slice boundary and MB boundary information could also be acquired from VPU. This informative encoder output information is generally placed on Parameter Buffer with pre-defined formats. So application can read out this information directly from the given Parameter Buffer using the base address of each data structure.

VPU API provides a function for getting output results of picture encoder, vpu_EncGetOutputInfo(), which has a output data structure including the following information:

- Start address of encoded picture and its size,
- Number of Sliced in the encoded picture,
- Slice boundary information in encoded bitstream,
- MB boundary information in encoded bitstream,
- Application-specific information for packetization.

Some packetization scheme such as Realtime Transfer Protocol (RTP) requires some different internal information of encoded picture depending on the codec standard. Slice information is very useful for some packet-based application which has some limitations of slice start in video packet. It could be also useful for implementing slice re-ordering on application side such as Arbitrary Slice Ordering (ASO) in H.264 standard.

In VPU API, we adopted a constraint on using this pair of encoder initiation function and encoder result acquisition. When using VPU API, application should always use these two functions as a pair. This means that without calling the result acquisition function, vpu_EncGetOutputInfo(), the next picture encoding operation will not be initiated by calling vpu_EncStartOneFrame() (Any other VPU commands will not be allowed unless application calls vpu_EncGetOutputInfo() after completion of picture encoding operation) This constraint will be very useful to protect the encoded results from overwritten from other thread by mistake in multiinstance environment. So application should regard vpu_EncGetOutputInfo() function as a releasing command of VPU from current picture encoding operation.

4.2.4 Terminating an Encoder Instance

When application is going to finish encoder operation and terminate an encoder instance, application just release the handle of this instance and let VPU know that this instance will be terminated by giving SEQ_END command to VPU. It can be done simply by calling vpu_EncClose() function.
4.2.5 Dynamic Configuration Commands

While running picture encoding operation sequentially, sometimes application needs to give a certain special commands to VPU such as rotating input pictures before encoding, insertion of high layer header syntaxes, etc. The VPU API provides a set of commands to support this kind of special requests from host application.

- Rotation and mirroring of source frame before encoding,
- Extract Sequence Parameter Set (SPS) and Picture Parameter Set (PPS) for external usage
- Insert high layer header syntaxes such as VOS, VO and VOL header in MPEG-4, and SPS and PPS in H.264,
- Set Search RAM base address
- Change encoder parameters such as bitrate, framerate, GOP number, slice mode and so on dynamically between picture encoding operations.

More special purpose commands will be available for providing rich set of user-friendly functionalities.

4.3 Decoder Control

4.3.1 Creating a Decoder Instance

After initialization of VPU, the first step to run decoder operation should be creation of a decoder instance, and acquisition of a handle for specifying that decoder instance. It could be easily done by using a single API function called `vpu_DecOpen()`.

When creating a new decoder instance, application should specify the internal features of this decoder instance through `DecOpenParam` structure. This structure includes the following information about the new decoder instance:

- Bitstream buffer address & size
  Physical address of bitstream buffer start address and its size
- Codec standard
  Video codec standard such as H.263, MPEG-4 or H.264
- MPEG-4 deblocking filter enable
  Enable or disable MPEG-4 deblocking filter option,
- qpReport
  Informative output data of Qstep values of each MB for external usage,
- reorderEnable
  Enable or disable H.264 display reordering option, which will be ignored in other codec standards.
- File play mode enable and picture size information
  Enable or disable frame-based streaming option for local file play mode. Picture size information has no meaning currently. It is just added for future use.
• SPS/PPS RBSP save buffer address & size

Physical address and size of buffer for save SPS and PPS.

In case of decoder, most information will be acquired from input stream itself, so required parameters for creating a decoder instance are not so many as required parameters for creating an encoder instance.

While creating a decoder instance for H.264 decoder, application should decide whether application uses display re-ordering option in H.264. Display re-ordering generally requires much more decoder buffers, much longer delay, and some complex constraints in decoder operations. Since display re-ordering is not actually used in practice even though display re-ordering option is enabled on the baseline profile stream, application could ignore this option with no big problems for reducing decoder delays, frame buffer size, etc.

A VPU API function, `vpu_DecOpen()`, does not require any operations on VPU side but declares all the internal parameters to be used in later stage as well as bitstream buffer information.

### 4.3.2 Configuring VPU for Decoder Instance

#### 4.3.2.1 Feeding bitstream into stream Buffer

In case of decoder, sequence initialization performs parsing of high level header syntaxes such as VOS/VO/VOL in MPEG-4 and SPS/PPS in H.264 for reading out decoder configurations. So in order to start sequence initialization, application should fill the decoder stream buffers with enough bitstream. In some applications, host applications could not guarantee that those kinds of header syntaxes are placed at the beginning of bitstream. In this case, until VPU successfully gets all the required information from input stream, application keep on feeding input data stream to decoder bitstream buffer.

In order to feeding input bitstream, host application should know the available space in bitstream buffer. It could be easily determined by using read pointer, write pointer and stream buffer size because stream buffer is working as a ring-buffer. Getting the available space in stream buffer, application can directly download decoder input stream to bitstream buffer. And after finishing stream download, application should inform the amount of downloaded stream by updating stream write pointer.

VPU API provides an updated API function to get stream read pointer, write pointer and available space by one function call, `vpu_DecGetBitstreamBuffer()`. And updating write pointer could also be done easily by using an API function, `vpu_DecUpdateBitstreamBuffer()`.

#### 4.3.2.2 Sequence Initialization

After creating a new instance and feeding input bitstream to stream buffer, application can give a command of DEC_SEQ_INIT to VPU in order to get decoder configuration information from bitstream. After parsing header syntaxes, decoder will return crucial information about decoder configuration including:
How to Control VPU

- Picture size
  
  Picture width and height,

- Frame rate

  Decoder frame rate

- Minimum number of Frame Buffers,

- MPEG-4 option information

  Enabled or disabled MPEG-4 error resilience options such as data partitioned or Reversible VLC as well as short video header mode,

- Frame buffer delay for display reordering

  The number of frame delays for supporting display reordering in H.264 decoder.

- Annex-J (Deblocking) option indication

  This flag indicates whether deblocking option of H.263 decoder is enabled or disabled (When external post-deblocking filter is used in H.263 case, this flag will be useful to avoid repetition of H.263 in-loop deblocking filter and external post-deblocking filter).

- Number of returned next decoded index after decoding one frame

  The number of returned indexes which will be used in next decoding after decoding one frame.

- Estimated slice save buffer sizes

  The size of slice save buffer. VPU will report two different sizes, recommended size and worst-case size.

The picture size acquired from bitstream might not be a multiple of 16x16. But in order to perform decoder operation properly, frame buffer size should be a multiple of 16x16. So the returned size will be modified to be a multiple of 16x16 after ceiling operation. By using the picture size and the minimum number of frame buffers, application should reserve frame buffers and provide them to VPU before starting picture decoding operation.

Frame buffer delay is an H.264-specific parameter for supporting display reordering. If application decide to support display reordering and reordering requires 5 additional frame buffers, for example, then the first display output will come out from decoder after decoding 6-th frame. Theoretically, maximum 16-frame delay would happen in display reordering.

VPU API provides an API function to handle this DEC_SEQ_INIT operations, vpu_DecGetInitialInfo(). Completion of this function will be signaled by a dedicated interrupt or polling BusyFlag.

An important issue in SEQ_INIT operation is error-handling because any errors in high layer header syntaxes will cause severe problems in decoding operations. Generally lots of marker bits are added into these header syntaxes in order to help error detection. In case that header syntaxes included in the stream have crucial errors, or header syntaxes are not received for a long time, VPU can be stuck on this task and no other instances would run on VPU at all. So VPU API provides a special function which could be used in this problematic situation only, called vpu_SetSeqInitEsc(). When this function is called and stream buffer is empty, VPU automatically terminates SEQ_INIT operation.
Then host application could decide whether closing this instance, retrying SEQ_INIT after running a different codec instance. After escaping from this situation, it is highly recommend to reset the internal ESCAPE flag by calling \texttt{vpu\_SetSeqInitEsc()} function again.\textcolor{red}{(This flag affects all the decoder instances performing DEC_SEQ_INIT operation. So this flag should be reset after escaping from stuck-on condition for safety)}

\subsection*{4.3.2.3 Registering Frame Buffers}

This configuring process will be finished by registering frame buffers to VPU for its future picture decoding operation. In this final stage of configuration, the returned parameter from \texttt{vpu\_DecGetInitialInfo()}, “the minimum number of frame buffer”, will have a very important meaning. This parameter means that application should reserve at least the same number of frame buffers to VPU for proper decoding operation.

One more option is available in MPEG-4 decoder case because of its deblocking filter option. In contrast to H.264 in-loop deblocking, MPEG-4 deblocking filter is a postfiltering operation. So application should declare a frame buffer for storing deblocking filtered output. This operation could be done by using \texttt{vpu\_DecGiveCommand()} with \texttt{SET\_DEBLOCKING\_OUTPUT} command.

In addition to registering frame buffers to VPU, registering slice save buffer is added to this step. The guideline of recommended buffer size will be given while calling \texttt{VPU\_DecGetInitialInfo()}.

\subsection*{4.3.3 Running Picture Decoder On VPU}

\subsubsection*{4.3.3.1 Initiating Picture Decoding}

When activating picture decoding operation, application should provide the following information to VPU:

- **Pre-scan Enable**
  
  Enable or disable pre-scan option for checking whether full picture stream exists in stream buffer

- **Pre-scan Mode**
  
  Specify decoder operation mode after pre-scan,

- **I-Frame Search Enable**
  
  Enable or disable I-(IDR in H.264 case) frame search option

- **Frame Skip Mode**
  
  Enable or disable skipping bitstream for the next frame decoding

- **isDispOrderBuf**
  
  Enable or disable the next display output without decoding

- **picStreamBufferAddr and picStartByteOffset**
  
  Start address of picture stream buffer to be decoded in file play mode and the byte offset of the actual start bytes of picture data
After providing all these parameters to VPU, application can start picture decoding operation by sending DEC_PIC_RUN command.

**Pre-scan is a special option for scanning the bitstream buffer to check whether full picture stream exists in stream buffer.** In general, by using this option, application can figure out whether bitstream empty and decoder hanging will happen or not before running actual decoder operation. When this option is enabled and there is no full picture stream in decoder buffer, DEC_PIC_RUN command will not initiate picture decoding operation and returns immediately. Then application could decide whether retrying picture decoding after feeding more bitstream or handling other tasks for a while.

Pre-scan mode is also given as an option for general usage of pre-scan operation. When this flag is set to 0 and there are at least one full picture stream in stream buffer, decoder operation will be automatically initiated. On the contrary, when this flag is set to 1, DEC_PIC_RUN command will return immediately with a return code representing whether a full picture stream exists or not. In this case, no picture decoding will be initiated. **In order to run picture decoding in this case, application should reset this flag by 0 and re-send DEC_PIC_RUN command.**

When display re-ordering in H.264 is enabled, the first decoded output to be displayed will only be available after decoding many frames. In order to avoid this problem, we added a constraint on H.264 decoder that at the first time of picture decoding, decoder should fill all the re-ordering display buffers. That means, if frame buffer delay received from stream header is 5, h.264 decoder should decode 6 frames at once at the first DEC_PIC_RUN operation. Then, all the picture decoding will always provide a picture output to be displayed. In this scenario, pre-scan might cause some problems, because it is designed for the case of one picture decoding. **So when display reordering is enabled, we recommend that the first DEC_PIC_RUN is performed without pre-scan enable option.**

In order to support display re-ordering in H.264, a special parameter will also be used to flush out stored decoder output from display re-order buffer without picture decoding. This option is specially designed for flushing out decoded picture but not yet displayed at the end of decoding video sequence. When display re-ordering option is enabled, and the re-ordering frame buffer will store 5 decoded pictures, the first display output will be available after 6-th frame decoding. **So at the end of stream decoding, there will be 5 decoded pictures which are not displayed so far even though there are no more available bitstream to decode.** In this case, application might ignore these 5 non-displayed pictures or can display them by setting isDispReorderBuff parameter by 1 and sending DEC_PIC_RUN command again and again until VPU returns decoded picture index of -1.

In file play mode, decoder will check the end of picture stream by using chunkSize input. So it is necessary to read this chunk size from file format header for every frame processing. Sometimes there might be some empty chunks whose chunk size equals to zero. All these empty chunks should be removed in file format parser because it might cause improper operations in VPU.

VPU API provides an API for handling all these complex operations, **vpu_DecStartOneFrame()**. Similar to encoder case, this function just initiates picture decoding operation and returns when
picture decoding is started on VPU. Completion of picture decoding will be checked by using an external way.

4.3.3.2 Frame skipping option

When decoder error is detected, application might want to hide that corrupted decoder output. Even though error concealment is applied to that decoder output, some application would like to freeze display instead of showing corrupted picture. This output-hiding operation should go on until decoder meets the next I (or IDR) frame. Considering AV synchronization, skipping one frame could be a good way to hide a sequence of pictures without affecting audio decoding operation.

Frame skipping option is supported as an option of picture decoding command. As well as skipping enable or disable, skipping option of detecting I (or IDR in H.264)-frame could be chosen by application. So when error is detected during picture decoding and application would like to hide all these error-defected pictures, application can achieve it by using picture skipping option with enabling I-frame detection. By setting skipframeMode of DecParam with 1, application easily performs skipping of non-intra (or non-IDR) frames. While application enables one frame skipping by setting skipframeNum of DecParam with 1, pre-scan will be automatically enabled and therefore, frame skip result will be translated to pre-scan result. While doing one frame skip, application could detect the results of frame skipping by checking prescanresult of DecOutputInfo.

This frame skip feature could be used by application when system performance is temporarily degraded and video decoding is delayed a lot. In this case, it would be highly recommended for application to use I-(IDR in H.264 case) frame detect option. By using this option, application could only decode I-(or IDR) frame properly without displaying error-defected frame output.

Multi-frame skipping is also supported by setting skipframeNum of DecParam greater than 1. But multi-frame skipping is not recommended in normal usage because it may cause some problems in AV synchronization generally.

In file play mode, frame skipping could be easily achieved in application side by referring the file format header syntax. So it is not required to support this feature in frame-based streaming case. But in random access case, I-frame search option could be useful when keyframe information in file container is incorrect.

4.3.3.3 I-frame Search for random access and trick mode

When a media player application is designed, trick modes and random access could be so desirable features. In order to achieve these operations on application decoder should support a feature of searching I-frame in the middle of decoder bitstream.

I-frame search option is very easily done by setting iframeSearchEnable of DecParam (The same skipframeNum of DecParam will be used for specifying the skipped frame number in frame skipping and I-search. The meaning of this memember value will be somewhat different). The number of I-frames skipped will also be set by setting skipframeNum of DecParam. If skipframeNum = N, all the intermediate frame before the (N+1)-th next I-frame will be skipped at once. This multiple I-frame skipping might be used for high speed playback such as fast forward. By increasing the number N, application could achieve increase the speed of fast forward. This kind
of fast forward operation basically depends on the frequency of I-(IDR) frames in decoder input bitstream. So this type of trick mode could be applicable to applications specifying the maximum interval of between I-frames.

Random access is generally supported with a form of slide-bar in graphic user interface of player. For supporting this random access, I-(or IDR in H.264) frame search operation is also needed because decoding intermediate inter-frames will cause visual artifacts on displayed pictures. As well as I-frame search functionality, random access also requires a buffer-reset scheme not to cause unexpected artifacts in decoded output. Steps of random access for video decoder could be summarized as follows:

- Step 1: Freeze display and reset decoder bit-stream buffer
- Step 2: Reading the bitstream from the new file read pointer and transferring it into decoder
- Step 3: Enabling I-Search and run picture decoding operation
- Step 4: If buffer empty is signaled through interrupt, then feeding more bit stream and wait for decoding completion.
- Step 5: Else if decoding completion is detected, read decoder results and resume display.

Resetting bitstream buffer in Step 1 could be easily done by calling an API, `vpu_DecBitBufferFlush()`. And starting decoder operation with I-frame search could also be done by calling `vpu_DecStartOneFrame()` with `iframeSearchEnable` of DecParam set by 1. The number of skipped frame specified by `skipframeNum` of DecParam will be given by 1 in random access operation. When an interrupt of decoder completion or non-busy state of BIT processor is detected, this means I-frame is searched and decoded.

When application uses I-frame search option, generally decoder should skip lots of bits in decoder stream buffer. So in this case, pre-scan option could be meaningless to use with I-search simultaneously. In VPU firmware, therefore, pre-scan option will be automatically disabled and settings for pre-scan option will also be ignored. That means application should take care of stream buffer filling until the end of I-search operation. Bigger unit of stream unit will be recommended in this case. If not, too much stream buffer empty interrupt might happen from VPU side.

4.3.3.4 Decoder Stream Handling

When decoder stream buffer includes a full picture stream, then host application does not need to care about streaming in the middle of decoder operation. By using Prescan option, application can figure out the status of bitstream buffer in advance. So if there is no full picture stream buffer in stream buffer, application might feed more stream data to stream buffer and start picture decoding operation.

VPU API provides an updated API function to get stream read pointer, write pointer and available space by one function call, `vpu_DecGetBitstreamBuffer()`. Application can get exact information about available space on stream buffer by using this API and transfer a certain amount of stream data to stream buffer which should be less than or equal to the available size. When transferring the stream data, application should take care of the end of stream buffer in order to avoid unexpected data corruption. While transferring stream data to stream buffer and write pointer get equal to the stream
How to Control VPU

API&Software Programmer's User Guide

Chips&Media, Freescale Semiconductor

buffer end. **application should wrap around the writing pointer to the beginning of stream buffer and go on downloading in order to avoid data corruption.**

And updating write pointer could also be done easily by using an API function, vpu_DecUpdateBitstreamBuffer(). **Write pointer wrap-around and updating write pointer will be done internally in this API function by providing just the downloaded stream size.**

4.3.3.5 Completion of Picture Decoding

Picture decoder operation will took a certain amount of time, and application could go on other tasks while waiting for the completion of picture decoding operation such as display processing of the previously decoded output. Application can use two different type of scheme for detecting completion of picture decoding operation, polling a status register or interrupt signaling. When application is going to use a polling scheme, application just needs to check the BusyFlag Register of BIT processor. Calling vpu_IsBusy() will give the same result.

Interrupt signaling could be the most efficient way to check the completion of a given command. An interrupt signal for DEC_PIC_RUN command is mapped on bit [3] of Interrupt Enable Register. So application could easily know the completion of picture decoder operation from this dedicated interrupt signal from VPU.

4.3.3.6 Acquiring Decoder Results

Whenever a picture decoding is completed, host application can get the decoded output such as decoded frame index, number of error concealed MB’s, MB Qstep, Prescan result, etc. According to the input parameter setting of picture decoding, MB Qstep information could be available or not.

The decoded frame index is generally used to point to the frame buffer where decoded picture is stored. But if display reordering is enabled, then it means the frame buffer index to be displayed this time. At the end of sequence decoding under display re-ordering, this value will be equal to -1 in order to represent that there are no more frames in display reordering buffer. When frame skip mode enabled, decoder return -2 if decoder doesn’t have frame displayed before end of sequence. **At the end of sequence during frame skip under display re-ordering, -1 will be returned from VPU in order to distinguish frame skip end from decoder end. So when application receives buffer index of -1, application can terminate current decoder instance without any loss in picture display.**

The pre-scan result flag presents whether a full picture stream is included in the bitstream buffer before picture decoding. When this flag is equal to 0, that means decoding operation is not performed because there is no full picture stream in stream buffer. So if application enables pre-scan and sets pre-scan mode by 0 (decoding a picture when full picture stream exists), then application should check this output parameter first of all to check whether decoding operation is performed or not.

VPU API provides a function for getting output results of picture decoder, vpu_DecGetOutputInfo(). In VPU API, we adopted a constraint on using this pair of decoder initiation function and decoder result acquisition. When using VPU API, application should always use these two functions as a pair. This means that without calling the result acquisition function, vpu_DecGetOutputInfo(), the next picture decoding operation will not be initiated by calling vpu_DecStartOneFrame(). This constraint will be very useful to protect the decoded results from
overwritten from other thread by mistake in multi-instance environment. So application should regard
vpu_DecGetOutputInfo() function as a releasing command of VPU from current picture decoding
operation.

• Reading display output from VPU decoder

The display frame index, indexFrameDisplay, is used to represent the frame buffer number where
display output picture is stored. It will always mean the frame buffer index to be displayed this time,
and it could be different from decoded picture index when display ordering control is enabled such as
display reordering of H.264.

At the beginning of sequence decoding, even after decoding several frames, there would be no display
output from decoder because of the order of display. In H.264 reorder case, the first display output can
be coming out after 17-th frame decoding in worst case. So sometimes, there would be no proper
display buffer index because of this reorder scheme. In this case, VPU decoder will return a negative
frame buffer index of indexFrameDisplay. It could be -3 or -2 depending on frame skip option. Only
at the end of sequence decoding, this value will be equal to -1. So when application receives buffer
index of -1, application can terminate current decoder instance without any loss in picture display. In
summary, host application can easily figure out the display output status and end of decoding sequence
as follows:

— Non-negative value of indexFrameDisplay

The output index value will point out the frame buffer index of display output
— indexFrameDisplay = -1

This means there would be no more display output when stream end is signaled to VPU. That is,
this means the end of sequence decoding.
— indexFrameDisplay = -2

This means there would be no display output temporarily because of the frame-skip option by
host application.
— indexFrameDisplay = -3

This means there would be no display output temporarily even without any action by host
application. In principle, this situation can happen when IDR picture is received under
display-reordering case in H.264 decoder.

• Reading decoded output from VPU decoder

The decoded frame index, indexFrameDecoded, is an optional output to host application. This index
is used to represent the frame buffer number where decoded picture is stored. Without no exceptions,
host application do not need to care about this index. The display index, indexFrameDisplay, would
be enough to handle the output of VPU decoder.
When display ordering control is enabled for H.264 display reordering, at the end of sequence decoding, host application needs to flush out decoded frames for display. During this flushing operation, no actual decoding operations will be performed. Under this situation, this value will be equal to -1 (0xFFFF) in order to represent that there is no decoded frame this time. This negative decoded index will also be used when picture decoding is skipped because of skip option or picture header error.

- Reading pre-scan result

The pre-scan result flag presents whether a full picture stream is included in the bitstream buffer before picture decoding. When this flag is equal to 0, that means decoding operation is not performed because there is no full picture stream in stream buffer. So if application enables pre-scan and sets pre-scan mode by 0 (decoding a picture when full picture stream exists), then application should check this output parameter first of all to check whether decoding operation is performed or not.

When pre-scan result is 0 and stream buffer is full, that means current stream buffer is too small to store a full picture stream in this time. So in order to avoid dead-lock, host application should disable pre-scan option and re-run picture decoding operation.

- Display Cropping in H.264

Display cropping option in H.264 forces host application to display part of frame buffers. The information about cropping window will be provided by SPS. In SPS, four offset values of cropping rectangles will be presented, and these four offset values will be given by picCropRect structure to host application. By using these four offset values, host application easily detect the position of target output window. When display cropping is off, the cropping window size will be 0.

- Reading frame index which will be used in next decoding

The next decoded frame index, indexNextFrameDecoded[3], is an optional output to host application. This indexes are used to represent the frame buffer index which will be used in next VPU_DecStartOneFrame() call. Application might not stop calling VPU_DecStartOneFrame() to protect display corruption, if some of these indexes are not displayed yet.

When display ordering control is enabled for H.264 display reordering, at the end of sequence decoding, host application needs to flush out decoded frames for display. During this flushing operation, no actual decoding operations will be performed. Under this situation, this value might be ignored.

- Reading lack of additional work buffer
VPU reports the status of PS (SPS/PPS) save buffer and slice save buffer after decodes one frame. If VPU reports lack of PS save buffer, VPU can not decode properly the remaining input stream. So it would be better to close current instance. If VPU reports lack of slice save buffer, VPU can choose either closing-and-reopening the current instance or continuing picture decoding regardless of display corruption until the next I-frame.

4.3.3.7 Escaping from decoder hanging

Even though pre-scan is used, it is still possible for application to experience decoder hanging because of stream error or lack of available stream at the end of sequence decoding. In the middle of picture decoding, decoder hanging will be signaled to application through decoder buffer empty interrupt if this interrupt is enabled, and application could avoid decoder hanging by putting more bitstream data to stream buffer. But in some extra-ordinary case and at the end of sequence decoding, this solution might not be applicable. In this case, application should avoid decoder hanging by means of garbage insertion to decoder stream buffer.

The format of effective garbage data is currently different from each codec standard and the size of effective garbage data could also be different. Based on the HW limitation and codec standard syntax, we recommend for application developers to use the following garbage data for each standard to escape from decoder hanging.

- H.263
  A 512 byte garbage data starting from a unique 32 bit value of 0x0000010B
- MPEG-4
  A 512 byte garbage data starting from a unique 32 bit value of 0x000001B6
- H.264
  A 512 byte garbage data starting from a unique 32 bit value of 0x0000010B

When decoder hanging happens at the end of input sequence during I-Search, inserting the garbage data described above will not be helpful at all. In that case, a set of different garbage data should be used as follows:

- H.263
  A 512 byte garbage data starting from a unique 32 bit value of 0x0000010B
- MPEG-4
  A 512 byte garbage data starting from a unique 32 bit value of 0x000001B1
- H.264
  A 512 byte garbage data starting from a unique 32 bit value of 0x000000FC

After escaping from decoder operation by using this garbage data, it is highly recommended to restart this decoder instance for safety.
4.3.4 Terminating an Decoder Instance

When application is going to finish decoder operation and terminate a decoder instance, application just release the handle of this instance and let VPU know that this instance will be terminated by giving SEQ_END command to VPU. It can be done simply by calling \texttt{vpu\_DecClose()} function.

4.3.5 Dynamic Configuration Commands

While running picture decoding operation sequentially, sometimes application needs to give a certain special commands to VPU. The VPU API provides a set of commands to support this kind of special requests from host application.

- Rotation and mirroring of output frame before decoding,
- Apply Sequence Parameter Set (SPS) and Picture Parameter Set (PPS) from external “out-of-band” protocol

More special purpose commands will be available for providing rich set of user-friendly functionalities.

4.4 Sample Applications

The section presents how to build and test the sample applications based on the API exported by VPU DLL.

There are 3 sample applications shipped with BSP:

- \textbf{ENCTEST}
  This application gives an example of how to use encoder API to control the VPU hardware to implement a encoder.

- \textbf{DECTEST}
  This application gives an example of how to use decoder API to control the VPU hardware to implement a decoder.

- \textbf{VPUDEMO}
  This application gives an example of how to use VPU API and other MX27 modules to implement the complicated program, which data path has CSI, Prp, VPU(Encoder), Buffer, VPU(Decoder), Pp and LCD. This demo also is an example of multi-instance using case.

4.4.1 System Requirement

In order to run the example applications, there are some requirements:

- OS image including eMMA, Camera, VPU and VGA panel supports (by modifying the code to use QVGA panel).
- Camera, which is supported by MX27 BSP, used for capturing the image for encoding.
- Raw bit stream data files used for decoding test
4.4.2 Build the WinCE6.0 Image and Example Applications

4.4.2.1 Build WinCE6.0 Image

The WinCE6.0 image used for VPU example applications must support the VGA panel. And in order to run the example application, the CE Target Control also be supported by the image.

The below is the steps of building the image with VGA support:
1. Change the default workspace to support DirectDraw display driver.
2. Select the Enable CE Target Control Support and Enable KITL check boxes.
3. Do sysgen as default image building.

4.4.2.2 Build and Test Example Applications

1. Click Build OS -> Open Release Directory to open the command prompt.
2. Run command "set wincrel=1" in command prompt window.
3. Change the current path to WINCE600\SUPPORT\MX27\APPS\VPUDEMO.
4. Build the application with "build -c" command.
5. Run test application using CE Target Control debugging command, such as "s dectest"

4.4.3 Run the Example Applications

There are four test case for running the example applications:

1. Single Decoding test:
   Run the dectest application by CE Target Control debugging command s dectest. The AVC bitstream File is played and video is displayed in VGA panel.

2. Single Encoding test:
   Run the enctest application by CE Target Control debugging command s enctest. The default test time is about 3000 frames captured by camera. After test application finished, you can copy it from WinCE device to workstation(laptop, etc), then play the encoded file using third party decoder.

3. Multi-process test:
   Start the Decoding test using s dectest, then run Encoding test using s enctest command. You can run with reverse sequence.

4. Loopback test:
   Loopback application is a test application with data path: CSI->Prp->VPU(Encoder)->Buffer->VPU(Decoder)->Pp->LCD. Using s vpudemo command to run the loopback test application. The images captured by camera are encoded and decoded again, then decoded images are displayed in VGA panel.