IMXVPUAPI

i.MX VPU Application Programming Interface Linux Reference Manual

Rev. LF6.1.55_2.2.0 — 15 December 2023 **Reference manual**

Document information

Information	Content
Keywords	i.MX, Linux, LF6.1.55_2.2.0
Abstract	This document provides the i.MX VPU API reference information for the Linux platform.

1 Overview

The i.MX SoC with Video Processing Units (VPU) supports the following three different VPUs:

- i.MX 6 Chips and Media VPU with a VPU library and firmware. This VPU has user space libraries that prepare IOCTL calls to the kernel VPU Chips and Media driver.
- i.MX 8M Hantro VPU with a VPU library and no firmware. This VPU has user space libraries that prepare IOCTL calls to the kernel VPU Hantro Driver.
- i.MX 8 and i.MX 8X Amphion VPU with firmware but no library. This VPU has no user space libraries and is interfaced with IOCTL calls to the kernel Video for Linux2 Driver or RPC communication.

In the Hantro and Chips and Media VPUs, the i.MX Multimedia framework provides a VPU wrapper interface that standardizes an API to all, even though each has different APIs to handle interactions with each VPU. This document describes those different interactions with more details coming in future releases.

1.1 VPU Wrapper

The VPU Wrapper library is a common interface to all the i.MX 6 VPUs and i.MX Hantro 8M VPUs for both Linux OS and Android platform. GStreamer delivers the header for VPU wrapper imx-gst1.0-plugin/ext-includes folder in the vpu_wrapper.h file. Samples for how to interface to the VPU wrapper are in the VPU plugin.

1.2 Hantro

The Hantro VPU on the i.MX 8M series of parts includes both a decoder and encoder. The user space library imx_vpu_hantro interfaces to the kernel Hantro VPU driver in drivers/mxc/hantro folders. The VPU Wrapper library interfaces to the Hantro library. Headers for the Hantro library are the hantrodec.h and hx280enc.h.

1.3 Amphion VPU RPC

The Amphion VPU hardware block on the i.MX 8Quad Max and i.MX 8QuadXPlus platforms uses dedicated Arm Cortex-M cores reserved for the VPU hardware decoder and encoder and is controlled by firmware running on the Arm Cortex-M cores. All APIs are exported to Arm cores through RPC protocol, which is implemented through the shared memory and MU interrupt. The kernel VPU driver resides in the folder drivers/mxc/vpu_malone and shows how to use the RPC interface to the firmware.

All RPC communication protocols are defined through some configuration parameters, commands, and callback event (e.g., message). Applications on Arm cores are responsible to send commands to decoder on the Cortex-M cores, such as start and stop. Decoder sends callback events to response application, such as start done, request frame buffers, and frame ready. All input and output parameters are transferred through the RPC shared memory. Decoder and encoder state machine should be maintained through one event handler implemented on Arm.

1.4 i.MX 6 VPU Overview

The i.MX 6 series Video Processing Unit (VPU) is a high performance multi-standard video decoder and encoder engine that performs multiple standard decoding and encoding operations. VPU codec is fully compliant with H.264 BP/MP/HP, VC-1 SP/MP/AP, MPEG-4 SP/ASP except GMC, DivX (Xvid), MPEG-1/2, VP8, AVS and MJPEG decoding and H.264, MPEG-4, H.263, and MJPG encoding. The VPU supports up to full HD 1920x1080 60i or 30p decoding and 1920x1088 encoding. It can encode or decode multiple video clips with multiple standards simultaneously. A block diagram of the i.MX 6 series VPU is shown in the figure below.

The VPU connects with the system through the 32-bit AMBA3 APB bus for system control and the 64-bit AMBA3 AXI for data throughput. The VPU also takes advantage of on-chip memories to achieve high performance.

Most video hardware blocks in the VPU are optimally designed for shared usage between different video standards which provides ultra low power and low gate count with powerful performance. As shown in the figure below, the VPU has a 16-bit DSP core, the BIT processor, which controls the internal video codec operations.

For simple and efficient control of the VPU by the host processor, the VPU provides a set of registers called the host interface registers. Most commands and responses between the host processor and the VPU are transmitted through the host interface registers. Stream data and some output picture data are directly accessed by the host processor and the VPU. For a more comprehensive way of controlling the VPU, a set of API functions is provided that includes all of the required operations from the host processor side.

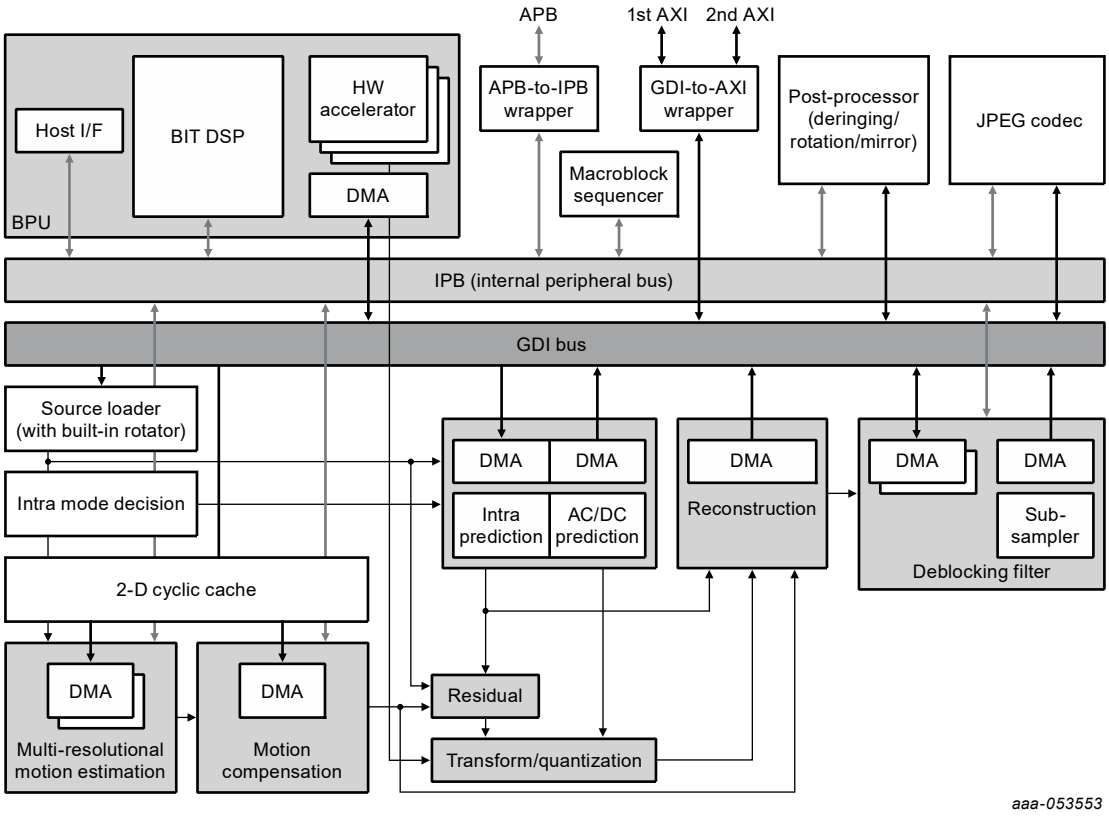

Figure 1. i.MX 6 VPU Block Diagram

2 VPU Wrapper Interface

VPU wrapper library provides encoding and decoding functions of streams for VPUs with library interfaces such as the Hantro VPU on the i.MX 8M family and the Chips and Media VPU used on the i.MX 6QuadPlus, 6Quad, 6Dual, and 6DualLite silicons. This is the API specification for the VPU wrapper library. The calling sequence of the API functions is also explained. The API is explicitly described in “vpu_wrapper.h”.

2.1 Data Types

2.1.1 Handle of VPU Encoder and Decoder

```
typedef void * VpuEncHandle;
typedef void * VpuDecHandle;
```

2.1.2 Enumerations

2.1.2.1 VpuEncRetCode and VpuDecRetCode

The following table lists the return values and descriptions for the encode and decoder API functions.

Return Value	Description
VPU_ENC_RET_SUCCESS/VPU_DEC_RET_SUCCESS	Success.
VPU_ENC_RET_FAILURE/VPU_DEC_RET_FAILURE	Failure.
VPU_ENC_RET_INVALID_PARAM/VPU_DEC_RET_INVALID_PARAM	Parameter is invalid.
VPU_ENC_RET_INVALID_HANDLE/VPU_DEC_RET_INVALID_HANDLE	Handle is invalid.
VPU_ENC_RET_INVALID_FRAME_BUFFER/VPU_DEC_RET_INVALID_FRAME_BUFFER	Frame buffer is invalid.
VPU_ENC_RET_INSUFFICIENT_FRAME_BUFFERS/VPU_DEC_RET_INSUFFICIENT_FRAME_BUFFERS	Frame buffers are insufficient.
VPU_ENC_RET_INVALID_STRIDE/VPU_DEC_RET_INVALID_STRIDE	Stride is invalid.
VPU_ENC_RET_WRONG_CALL_SEQUENCE/VPU_DEC_RET_WRONG_CALL_SEQUENCE	State of the object is not correct.
VPU_ENC_RET_FAILURE_TIMEOUT/VPU_DEC_RET_FAILURE_TIMEOUT	Waiting for hardware software to finish times out.

2.1.2.2 VpuDecRetCode

The following table lists the return values and descriptions for the decoder API functions.

VpuDecBufRetCode Return Value	Description
VPU_DEC_INPUT_NOT_USED	Input data has been consumed.
VPU_DEC_INPUT_USED	Input data hasn’t been consumed.
VPU_DEC_OUTPUT_EOS	Received end of stream.
VPU_DEC_OUTPUT_DIS	Received one frame to output.
VPU_DEC_OUTPUT_NODIS	Received no frame to output.
VPU_DEC_OUTPUT_REPEAT	One frame is output repeatedly.
VPU_DEC_OUTPUT_DROPPED	Received one frame to drop.
VPU_DEC_OUTPUT_MOSAIC_DIS	Received one mosaic frame to output.
VPU_DEC_NO_ENOUGH_BUF	Not enough buffer to hold for output.
VPU_DEC_NO_ENOUGH_INBUF	Not enough input buffer.
VPU_DEC_INIT_OK	Initialization of decoding is ok.
VPU_DEC_SKIP	Skip to decode one frame.
VPU_DEC_ONE_FRM_CONSUMED	One frame has been decoded.
VPU_DEC_RESOLUTION_CHANGED	Resolution of the frame is changed.
VPU_DEC_FLUSH	Flush the decoder.

2.1.2.3 VpuEncRetCode

The following table lists the return values and descriptions for the encode API functions.

VpuEncBufRetCode Return Value	Description
VPU_ENC_INPUT_NOT_USED	Input data has been consumed.
VPU_ENC_INPUT_USED	Input data hasn't been consumed.
VPU_ENC_OUTPUT_SEQHEADER	Sequence header (for H.264: SPS/PPS).
VPU_ENC_OUTPUT_DIS	Got one frame to output.
VPU_ENC_OUTPUT_NODIS	Got no frame to output.

2.1.2.4 VpuDecCapability

The following table lists the capabilities sof the decoder.

VPUDecCapability Value	Description
VPU_DEC_CAP_FILEMODE = 0	File mode is supported? 0: not; 1: yes
VPU_DEC_CAP_TILE	Tile format is supported? 0: not; 1: yes
VPU_DEC_CAP_FRAMESIZE	Reporting frame size? 0: not; 1: yes
VPU_DEC_CAP_RESOLUTION_CHANGE	Resolution change notification? 0: not; 1: yes

2.1.2.5 VpuDecConfig

Specifies the configuration types of decoder.

VpuDecConfig Value	Description
VPU_DEC_CONF_SKIPMODE = 0	Parameter value: • VPU_DEC_SKIPNONE (default) • VPU_DEC_SKIPPB • VPU_DEC_SKIPB • VPU_DEC_SKIPALL • VPU_DEC_ISEARCH
VPU_DEC_CONF_INPUTTYPE	Parameter value: • VPU_DEC_IN_NOMAL: normal (default); • VPU_DEC_IN_KICK: kick -- input data/size in VPU_ DecDecodeBuf() will be ignored; • VPU_DEC_IN_DRAIN: drain -- stream reach end, and input data/size in VPU_DecDecodeBuf() will be ignored.
VPU_DEC_CONF_BUFDELAY	For stream mode. The parameter represents buffer size (unit: bytes). Buffer size==0 indicates no delay.
VPU_DEC_CONF_INIT_CNT_THRESHOLD	At seqinit stage. VPU reports an error if data count reaches the threshold.
VPU_DEC_CONF_ENABLE_TILED	Configure output frame to tiled after parsing sequence header and before registering the frame buffer.

2.1.2.6 VpuEncConfig

Specifies the configuration types of encoder.

VpuEncConfig Value	Description
VPU_ENC_CONF_NONE = 0	None.
VPU_ENC_CONF_BIT_RATE	Set bit rate (unit: kbps).
VPU_ENC_CONF_INTRA_REFRESH	Intra refresh: minimum number of macroblocks to refresh in a frame.
VPU_ENC_CONF_ENA_SPSPPS_IDR	Some muxers may ignore the sequence or configure data, so SPS/PPS is needed for every IDR frame, including the first IDR.
VPU_ENC_CONF_RC_INTRA_QP	Intra QP value.
VPU_ENC_CONF_INTRA_REFRESH_MODE	Intra refresh mode: 0: normal; 1: cyclic.

2.1.2.7 VpuMemType

Specifies the memory type of vpu.

VpuMemType Value	Description
VPU_MEM_VIRT = 0	0 for virtual memory
VPU_MEM_PHY = 1	1 for physical continuous memory

2.1.2.8 VpuDecErrInfo

Specifies the type of error information during decode.

VpuDecErrInfo Value	Description
VPU_DEC_ERR_UNFOUND = 0	None.
VPU_DEC_ERR_NOT_SUPPORTED	The profile/level/features/... outranges the VPU's capability.
VPU_DEC_ERR_CORRUPT	Some syntax errors are detected.

2.1.2.9 VpuPicType

Specifies the picture types of VPU.

VpuPicType Value	Description
VPU_I_PIC = 0	I frame or I slice (H.264).
VPU_P_PIC	P frame or P slice (H.264).
VPU_B_PIC	B frame or B slice H.264).
VPU_IDR_PIC	IDR frame (H.264).
VPU_BI_PIC	BI frame (VC1)
VPU_SKIP_PIC	Skipped frame (VC1).
VPU_UNKNOWN_PIC	Reserved.

2.1.2.10 VpuFieldType

Specifies the field type of VPU.

VpuFieldType Value	Description
VPU_FIELD_NONE = 0	Frame
VPU_FIELD_Top	Only top field.
VPU_FIELD_BOTTOM	Only bottom field.
VPU_FIELD_TB	Top field + Bottom field.
VPU_FIELD_BT	Bottom field + Top field.
VPU_FIELD_UNKNOWN	Reserved.

2.1.2.11 VpuType

Specifies the type of VPU.

```
typedef enum {
    VPU_TYPE_UNKNOWN = 0,
    VPU_TYPE_CHIPSMEDIA,
    VPU_TYPE_MALONE,
    VPU_TYPE_HANTRO,
} VpuType;
```

2.1.2.12 VpuCodStd

Specifies the video type.

```
typedef enum {
  VPU_V_MPEG4 = 0,
  VPU_V_DIVX3,
  VPU_V_DIVX4,
  VPU_V_DIVX56,
  VPU_V_XVID,
  VPU_V_H263,
  VPU_V_AVC,
  VPU_V_AVC_MVC,
  VPU_V_VC1,
  VPU_V_VC1_AP,
  VPU_V_MPEG2,
  VPU_V_RV,
  VPU_V_MJPG,
  VPU_V_AVS,
  VPU_V_VP8,
  VPU_V_VP9,
  VPU_V_HEVC,
  VPU_V_SORENSON,
  VPU_V_VP6,
  VPU_V_WEBP,
} VpuCodStd;
```

2.1.2.13 VpuDecSkipMode

Specifies the skip mode when decoding.

```
typedef enum {
  VPU_DEC_SKIPNONE=0,
  VPU_DEC_SKIPPB,
  VPU_DEC_SKIPB,
  VPU_DEC_SKIPALL,
  VPU_DEC_ISEARCH,        /*only decode IDR*/
}VpuDecSkipMode;
```

2.1.2.14 VpuDecInputType

Specifies the input type of decoder.

```
typedef enum {
  VPU_DEC_IN_NORMAL=0,
  VPU_DEC_IN_KICK,
  VPU_DEC_IN_DRAIN,
}VpuDecInputType;
```

2.1.2.15 VpuColorFormat

Specifies the color format of video.

```
typedef enum
{
  VPU_COLOR_420=0,
  VPU_COLOR_422H=1,
  VPU_COLOR_422V=2,
  VPU_COLOR_444=3,
  VPU_COLOR_400=4,
  VPU_COLOR_422YUYV=13,
  VPU_COLOR_422UYVY=14,
  VPU_COLOR_ARGB8888=15,
  VPU_COLOR_BGRA8888=16,
  VPU_COLOR_RGB565=17,
  VPU_COLOR_RGB555=18,
  VPU_COLOR_BGR565=19,
}VpuColorFormat;
```

2.1.2.16 VpuEncMirrorDirection

Specifies the mirror direction of encoder.

```
typedef enum {
  VPU_ENC_MIRDIR_NONE,
  VPU_ENC_MIRDIR_VER,
  VPU_ENC_MIRDIR_HOR,
  VPU_ENC_MIRDIR_HOR_VER
```

```
} VpuEncMirrorDirection;
```

2.1.2.17 VpuMemDescType

Specifies the memory type.

```
typedef enum{
    VPU_MEM_DESC_NORMAL = 0,
    VPU_MEM_DESC_SECURE = 1,
}VpuMemDescType;
```

2.1.3 Enumerations

2.1.3.1 VpuMemSubBlockInfo

A struct that contains the information of a subblock.

Members	Type	Description
nAlignment	int	Alignment limitation.
nSize	int	Size of memory length.
MemType	VpuMemType	Flag to indicate Static, scratch, or output data memory.
pVirtAddr	unsigned char *	Virtual address of the pointer to the base memory.
pPhyAddr	unsigned char *	Physical address of the pointer to the base memory.
nReserved[3]	int	Reserved for future extension.

2.1.3.2 VpuMemInfo

A struct that contains memory information of all subblocks.

Members	Type	Description
nSubBlockNum	int	Number of subblocks.
MemSubBlock[VPU_DEC_MAX_NUM_MEM_REQS]	VpuMemSubBlockInfo	VpuMemSubBlockInfo struct that contains subblock information.

2.1.3.3 VpuVersionInfo

A struct that contains memory information of all subblocks.

Members	Type	Description
nFwMajor	int	Firmware major version.
nFwMinor	int	Firmware minor version.
nFwRelease	int	Firmware release version.
nFwCode	int	Firmware code version.

Members	Type	Description
nLibMajor	int	Library major version
nLibMinor	int	Library minor version.
nLibRelease	int	Library release version.
nReserved	int	Reserved for future extension.

2.1.3.4 VpuWrapperVersionInfo

A struct that contains vpu wrapper version information.

Members	Type	Description
nMajor	int	Major Version
nMinor	int	Minor Version
nRelease	int	Release Version
pBinary	char *	Version information specified by user.
nReserved[4]	int	Reserved for future extension.

2.1.3.5 VpuFrameBuffer

A struct that contains the information of frame buffer in VPU.

Members	Type	Description
nStrideY	unsigned int	Luma stride information.
nStrideC	unsigned int	Chroma stride information.
pbufY	unsigned char *	Physical address of luma frame pointer or top field pointer.
pbufCb	unsigned char *	Physical address of chroma frame pointer or top field pointer.
pbufCr	unsigned char *	-
pbufMvCol	unsigned char *	-
pbufY_tilebot	unsigned char *	For field tile: physical address of luma bottom pointer.
pbufCb_tilebot	unsigned char *	For field tile: physical address of chroma bottom pointer.
pbufVirtY	unsigned char *	Virtual address of luma frame pointer or top field pointer.
pbufVirtCb	unsigned char *	Virtual address of chroma frame pointer or top field pointer.
pbufVirtCr	unsigned char *	-
pbufVirtMvCol	unsigned char *	-
pbufVirtY_tilebot	unsigned char *	For field tile: virtual address of luma bottom pointer.
pbufVirtCb_tilebot	unsigned char *	For field tile: virtual address of chroma bottom pointer.

Members	Type	Description
nReserved[5]	int	Reserved for future extension.
pPrivate	void *	Reserved for future special extension.

2.1.3.6 VpuRect

A struct that contains the image information

Members	Type	Description
nLeft	unsigned int	Size of the image left.
nTop	unsigned int	Size of the image top.
nRight	unsigned int	Size of the image right.
nBottom	unsigned int	Size of the image bottom.

2.1.3.7 VpuHDR10Meta

A struct that contains the meta data of video hdr10.

Members	Type	Description
redPrimary[2]	unsigned int	-
greenPrimary[2]	unsigned int	-
bluePrimary[2]	unsigned int	-
whitePoint[2]	unsigned int	-
maxMasteringLuminance	unsigned int	-
minMasteringLuminance	unsigned int	-
maxContentLightLevel	unsigned int	-
maxFrameAverageLightLevel	unsigned int	-

2.1.3.8 VpuColourDesc

A struct that contains color description.

Members	Type	Description
colourPrimaries	unsigned int	-
transferCharacteristics	unsigned int	-
matrixCoeffs	unsigned int	-
fullRange	unsigned int	-

2.1.3.9 VpuChromaLocInfo

A struct that contains chroma description.

Members	Type	Description
chromaSampleLocTypeTopField	unsigned int	-

Members	Type	Description
chromaSampleLocTypeBottomField	unsigned int	-

2.1.3.10 VpuDecInitInfo

A struct that contains the initial information of decoder.

Members	Type	Description
nPicWidth	int	Aligned width of image.
nPicHeight	int	Aligned height of image.
nFrameRateRes	int	Numerator of framerate.
nFrameRateDiv	int	Denominator of framerate.
PicCropRect	VpuRect	Struct that contains crop information of image.
nMinFrameBufferCount	int	Minimum frame buffer count in VPU.
nMjpgSourceFormat	int	Source color format of jpeg
nInterlace	int	Whether video is interlaced.
nQ16ShiftWidthDivHeightRatio	unsigned int	Fixed point for width/height: 1: 0x10000; 0.5: 0x8000; ...
nConsumedByte	int	Reserved to record sequence length: value -1 indicate unknow.
nAddressAlignment	int	Address alignment for Y/Cb/Cr (unit: bytes).
nFrameSize	int	Hantro video decoder append DMV and compression table in pixel buffer.
nBitDepth	int	Bit depth of video.
nReserved[3]	int	Reserved for future extension.
pSpecialInfo	void *	Reserved for future special extension.
hasColorDesc	int	Whether has color description.
hasHdr10Meta	int	Whether has hdr10 meta data.
Hdr10Meta	VpuHDR10Meta	Hdr10Meta struct which contains hdr10 meta data.
ColourDesc	VpuColourDesc	ColourDesc struct which contains color description.
ChromaLocInfo	VpuChromaLocInfo	VpuChromaLocInfo struct which contains chroma information.

2.1.3.11 VpuFrameExtInfo

A struct that contains the extended information of frame.

Members	Type	Description
nFrmWidth	int	Width of image.
nFrmHeight	int	Height of image.

Members	Type	Description
FrmCropRect	VpuRect	VpuRect struct that contains the crop information of image.
nQ16ShiftWidthDivHeightRatio	unsigned int	Fixed point for width/height: 1: 0x10000; 0.5: 0x8000;...
rfc_luma_offset	int	Luma offset.
rfc_chroma_offset	int	Chroma offset.
nReserved[7]	int	Reserved for future extension.

2.1.3.12 VpuDecOutFrameInfo

A struct that contains information of output frame when decoding.

Members	Type	Description
pDisplayFrameBuf	VpuFrameBuffer *	Pointer to VpuFrameBuffer struct which contains the display information of the frame.
ePicType	VpuPicType	Type of frame.
eFieldType	VpuFieldType	Field type of VPU.
nMVCViewID	int	Extended info: support dynamic resolution, ...
pExtInfo	VpuFrameExtInfo *	Luma offset.
nReserved[2]	int	Reserved for future extension.
pPrivate	void *	Reserved for future special extension.

2.1.3.13 VpuCodecData

A struct that contains the codec data information.

Members	Type	Description
pData	unsigned char *	Codec data virtual address.
nSize	unsigned int	Codec data length.

2.1.3.14 VpuRBufferNode

A struct that contains the information of a buffer in vpu.

Members	Type	Description
pPhyAddr	unsigned char *	Buffer physical base address.
pVirAddr	unsigned char *	Buffer virtual base address.
nSize	unsigned int	Length of data.
sCodecData	VpuCodecData	VpuCodecData struct that contains codec data information.
nReserved[2]	int	Reserved for future extension.
pPrivate	void *	Reserved for future special extension

2.1.3.15 VpuMemDesc

A struct that contains the description of memory.

Members	Type	Description
nSize	int	Requested memory size.
pPhyAddr	unsigned long	Physical memory address allocated.
nCpuAddr	unsigned long	CPU address for system free usage.
nVirtAddr	unsigned long	Virtual user space address.
nType	VpuMemDescType	Type of memory.
nReserved[3]	int	Reserved for future extension.

2.1.3.16 VpuDecFrameLengthInfo

A struct that contains the information of the frame length in vpu.

Members	Type	Description
pFrame	VpuFrameBuffer *	Point to the frame buffer which contains the information of frame buffer.
nStuffLength	int	Stuff data length ahead of frame.
nFrameLength	int	Length of the frame.
nReserved[5]	int	Reserved for recording other information.

2.1.3.17 VpuEncInitInfo

A struct that contains the initial information of encoder.

Members	Type	Description
nMinFrameBufferCount	int	Minimum frame buffer count in VPU.
nAddressAlignment	int	Address alignment for Y/Cb/Cr (unit: bytes).
eType	VpuType	Type of VPUs.

2.1.3.18 VpuEncOpenParamSimp

A struct that contains the simple input parameters of decoder.

Members	Type	Description
eFormat	VpuCodStd	Type of video.
nPicWidth	int	Width of the encoded image.
nPicHeight	int	Height of the encoded image.
nRotAngle	int	Rotate angle of the image.
nFrameRate	int	The framerate of the output frame.
nBitRate	int	Bit rate of the output frame.

Members	Type	Description
nGOPSize	int	Number of pictures in one GOP.
nIntraRefresh	int	Intra macro block numbers.
nIntraQP	int	Qp values, 0: auto, >0: qp value.
nChromaInterleave	int	Should be set to 1 when (nMapType! =0).
sMirror	VpuEncMirrorDirection	Mirror direction of the encoder.
nMapType	int	Frame buffer: 0--linear ; 1--frame tile; 2--field tile.
nLinear2TiledEnable	int	Valid when (nMapType!=0): 0--tile input; 1--yuv input.
eColorFormat	VpuColorFormat	Color format of video.
nIsAvcc	int	Used for H.264 data format, 0: byte stream ; 1: avcc format.
nReserved[3]	int	Reserved for future extension.
pAppCxt	void *	Reserved for future extension.

2.1.3.19 VpuEncSliceMode

A struct that contains the slice information.

Members	Type	Description
sliceMode	int	The mode of slice.
sliceSizeMode	int	The mode of slice size.
sliceSize	int	Set the size of a slice.
nReserved	int	Reserved for future extension.

2.1.3.20 VpuEncOpenParam

A struct that contains input parameters of decoder.

Members	Type	Description
eFormat	VpuCodStd	Type of video.
nPicWidth	int	Width of the encoded image.
nPicHeight	int	Height of the encoded image.
nRotAngle	int	Rotate angle of the image.
nFrameRate	int	Framerate of the output frame.
nBitRate	int	Bit rate of the output frame.
nGOPSize	int	Number of pictures in one GOP.
nChromaInterleave	int	Should be set to 1 when (nMapType! =0).
sMirror	VpuEncMirrorDirection	Mirror direction of the encoder.

Members	Type	Description
nMapType	int	Frame buffer: 0--linear ; 1--frame tile; 2--field tile.
nLinear2TiledEnable	int	Valid when (nMapType!=0): 0--tile input; 1--yuv input.
eColorFormat	VpuColorFormat	Color format of video.
nUserQpMax	int	Maximum user qp value.
nUserQpMin	int	Minimum user qp value.
nUserQpMinEnable	int	-
nUserQpMaxEnable	int	-
nIntraRefresh	int	Intra macro block numbers.
nRcIntraQP	int	QP values, 0: auto, >0: QP value.
nUserGamma	int	
nRcIntervalMode	int	0: normal, 1: frame_level, 2: slice_level, 3: user defined Mb_level.
nMbInterval	int	Used when RcintervalMode is 3.
nAvcIntra16x16OnlyModeEnable	int	-
sliceMode	VpuEncSliceMode	A pointer to VpuEncSliceMode struct.
nInitialDelay	int	-
nVbvBufferSize	int	-
union { VpuEncMp4Param mp4Param; VpuEncH263Param h263Param; VpuEncAvcParam avcParam; } VpuEncStdParam;		-
nMESearchRange	int	3: 16x16, 2: 32x16, 1: 64x32, 0: 128x64, H.263 (Short Header: always 3.
nMEUseZeroPmv	int	0: PMV_ENABLE, 1: PMV_DISABLE.
IntraCostWeight	int	Additional weight of Intra Cost for mode decision to reduce Intra MB density.
nIsAvcc	int	Used for H.264 data format, 0: byte stream; 1: avcc format.
nReserved[8]	int	Reserved for future extension.
pAppCxt	void *	Reserved for future extension.

2.1.3.21 VpuEncEncParam

A struct that contains the input and output parameters of encoder.

Members	Type	Description
eFormat	VpuCodStd	Type of video.
nPicWidth	int	Width of the encoded image.
nPicHeight	int	Height of the encoded image.

Members	Type	Description
nFrameRate	int	Framerate of the output frame.
nQuantParam	int	Quant parameter of the frame
nInPhyInput	unsigned long	Input buffer physical address.
nInVirtInput	unsigned long	Input buffer virtual address.
nInInputSize	int	Input buffer size.
nInPhyOutput	unsigned long	Output buffer physical address.
nInVirtOutput	unsigned long	Output buffer virtual address.
nInOutputBufLen	unsigned int	The left buffer size.
nForceIPicture	int	Whether to force the current frame as key frame.
nSkipPicture	int	Whether to skip current frame.
nEnableAutoSkip	int	Whether to enable auto skip.
eOutRetCode	VpuEncBufRetCode	Output state after encoding.
nOutOutputSize	int	Output buffer size.
pInFrame	VpuFrameBuffer *	Pointer to VpuFrameBuffer struct that contains the input frame information.
nReserved[2]	int	Reserved for future extension.
pPrivate	void *	Reserved for future extension.

2.2 Decoder API Functions

2.2.1 Decoder Open and Close

2.2.1.1 VPU_DecGetVersionInfo

```
VpuDecRetCode VPU_DecGetVersionInfo (
       VpuVersionInfo     * pOutVerInfo)
```

Description: Function to get the vpulib and firmware version.

Arguments

- pOutVerInfo [out] - Pointer to VpuVersionInfo struct where output parameters will be saved.

Return value

VPU_DEC_RET_SUCCESS

2.2.1.2 VPU_DecGetWrapperVersionInfo

```
VpuDecRetCode VPU_DecGetWrapperVersionInfo (
      VpuWrapperVersionInfo   * pOutVerInfo)
```

Description:

Function to get the VPU wrapper version.

Arguments

- pOutVerInfo [out] - Pointer to VpuWrapperVersionInfo struct where the output parameters will be saved.

Return value

VPU_DEC_RET_SUCCESS

2.2.1.3 VPU_DecGetInitialInfo

```
VpuDecRetCode VPU_DecGetInitialInfo (
  VpuDecHandle     InHandle,
  VpuDecInitInfo     * pOutInitInfo)
```

Description:

Function to get the initial information.

Arguments:

- InHandle [in/out] - Handle of VPU decoder.
- pOutInitInfo [out] - Pointer to VpuEncInitInfo struct where initial information is stored.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.1.4 VPU_DecConfig

```
VpuDecRetCode VPU_DecConfig (
  VpuDecHandle     InHandle,
  VpuDecConfig     InDecConf,
  void      * pInParam)
```

Description:

Function to set corresponding parameters of decoder according to InDecConf type.

Arguments:

- InDecConf [in] - Type of configuration to be set.
- pInParam [in] - Value that used to set the decoder according to InDecConf type.
- InHandle [out] - Handle of VPU decoder.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.1.5 VPU_DecOpen

```
VpuDecRetCode VPU_DecOpen (
  VpuDecHandle     * pOutHandle,
  VpuDecOpenParam   * pInParam
  VpuMemInfo    * pInMemInfo)
```

Description:

Function to open new VPU handle.

Arguments:

- pInParam [in] - Pointer to VpuDecOpenParam struct which contains input parameters of the decoder.
- pInMemInfo [in] - Pointer to VpuMemInfo struct which contains the memory information.
- pOutHandle [out] - Handle of VPU decoder.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.1.6 VPU_DecGetCapability

```
VpuDecRetCode VPU_DecGetCapability (
  VpuDecHandle     InHandle,
  VpuDecCapability   eInCapability,
  int      * pOutCapbility)
```

Description:

Function to get capability of the input eInCapability. If eInCapability is supported, pOutCapbility will be set to 1.

Arguments:

- InHandle [in] - Handle of VPU decoder.
- eInCapability [in] - Type of configuration to be set.
- pOutCapbility [out] - Set to be 1 if eInCapability type is supported.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.1.7 VPU_DecDisCapability

```
VpuDecRetCode VPU_DecDisCapability (
  VpuDecHandle     InHandle,
  VpuDecCapability   eInCapability)
```

Description:

Function to get capability of the input eInCapability. If eInCapability is supported, pOutCapbility will be set to 1.

Arguments:

- InHandle [in/out] - Handle of VPU decoder, corresponding parameters will be set according to eInCapability type.

- eInCapability [in]- Type of configuration to be set.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.1.8 VPU_DecGetErrInfo

```
VpuDecRetCode VPU_DecGetErrInfo (
  VpuDecHandle     InHandle,
  VpuDecErrInfo    * pErrInfo)
```

Description:

Function to get the error information.

Arguments:

- InHandle [in] - Handle of VPU decoder.
- pErrInfo [out] - Pointer to VpuDecErrInfo enumeration where contains type of error information.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.1.9 VPU_DecGetNumAvailableFrameBuffers

```
VpuDecRetCode VPU_DecGetNumAvailableFrameBuffers (
  VpuDecHandle     InHandle,
  int      * pOutBufNum)
```

Description:

Function to get the number of available frame buffers.

Arguments:

- InHandle [in] - Handle of VPU decoder.
- pOutBufNum [out] - Number of the available frame buffers.

Return value:

VPU_DEC_RET_SUCCESS

2.2.1.10 VPU_DecUnLoad

```
VpuDecRetCode VPU_DecUnLoad ()
```

Description:

Function to unload VPU.

Arguments:

None.

Return value:

VPU_DEC_RET_SUCCESS

2.2.1.11 VPU_DecReset

```
VpuDecRetCode VPU_DecReset (
  VpuDecHandle     InHandle)
```

Description:

Function to reset the handle of VPU decoder.

Arguments:

InHandle [in] - Handle of VPU decoder.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.1.12 VPU_DecClose

```
VpuDecRetCode VPU_DecClose (
  VpuDecHandle     InHandle)
```

Description:

Function to close the decoder.

Arguments:

InHandle [in] - Handle of VPU decoder.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.1.13 VPU_DecFlushAll

```
VpuDecRetCode VPU_DecFlushAll (
  VpuDecHandle     InHandle)
```

Description:

Function to flush the decoder.

Arguments:

InHandle [in] - Handle of VPU decoder.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.2 Decode

2.2.2.1 VPU_DecDecodeBuf

```
VpuDecRetCode VPU_DecDecodeBuf (
  VpuDecHandle     InHandle,
  VpuBufferNode    * pInData,
      int      * pOutBufRetCode)
```

Description:

Function to decode one frame. This function will see if there is decoded frame to be sent out, then process the input buffer of the current frame and decode it. The final decode state is saved in pOutBufRetCode.

Arguments:

- InHandle [in/out] - Handle of VPU decoder.
- pInData [in] - Pointer to VpuBufferNode struct where the information of input buffer is stored.
- pOutBufRetCode [out] - State of the input and output frame.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.2.2 VPU_DecGetOutputFrame

```
VpuDecRetCode VPU_DecGetOutputFrame (
  VpuDecHandle       InHandle,
  VpuDecOutFrameInfo     * pOutFrameInfo)
```

Description:

Function to get the next output frame information.

Arguments:

- InHandle [in/out] - VPU decode object.
- pOutFrameInfo [out] - Pointer to VpuDecOutFrameInfo struct where the output frame information is stored.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.2.3 VPU_DecGetConsumedFrameInfo

```
VpuDecRetCode VPU_DecGetConsumedFrameInfo (
  VpuDecHandle       InHandle,
  VpuDecFrameLengthInfo  * pOutFrameInfo)
```

Description:

Function to get the information of frame buffers in VPU.

Arguments:

- InHandle [in/out] - VPU decode object.
- pOutFrameInfo [out] - Pointer to VpuDecFrameLengthInfo struct where the information of frame buffers in VPU is stored.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.2.4 VPU_DecOutFrameDisplayed

```
VpuDecRetCode VPU_DecOutFrameDisplayed (
  VpuDecHandle     InHandle,
  VpuFrameBuffer  * pInFrameBuf)
```

Description:

Function to clear one display frame buffer.

Arguments:

- InHandle [in/out] - Handle of VPU decoder.
- pInFrameBuf [out] - Pointer to VpuFrameBuffer struct which contains the information of frame buffer to be cleared.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.3 Memory Query and Free

2.2.3.1 VPU_DecQueryMem

```
VpuDecRetCode VPU_DecQueryMem (
  VpuMemInfo    * pOutMemInfo)
```

Description:

Function to query output physical and virtual memory information to further allocate.

Arguments:

- pOutMemInfo [out] - Pointer to VpuMemInfo struct where the output memory information is stored.

Return value:

VPU_DEC_RET_SUCCESS

2.2.3.2 VPU_DecGetMem

```
VpuDecRetCode VPU_DecGetMem (
  VpuMemDesc    * pInOutMem)
```

Description:

Function to allocate memory.

Arguments:

- pInOutMem [in/out] - Pointer to VpuMemDesc struct where the memory information is stored.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.3.3 VPU_DecFreeMem

```
VpuDecRetCode VPU_DecFreeMem (
  VpuMemDesc    * pInMem)
```

Description:

Function to free memory.

Arguments:

- pInMem [in] - Pointer to VpuMemDesc struct where the memory information is stored.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.2.3.4 VPU_DecRegisterFrameBuffer

```
VpuDecRetCode VPU_DecRegisterFrameBuffer (
  VpuDecHandle     InHandle,
  VpuFrameBuffer   * pInFrameBufArray,
  int       nNum)
```

Description:

Function to register frame buffer.

Arguments:

- InHandle [in/out] - Handle of VPU encoder.
- pInFrameBufArray [in] - Pointer to VpuFrameBuffer struct where the information of VPU frame buffer is stored.

- nNum [in] - The number of VPU frame buffer count.

Return value:

VpuDecRetCode value: VPU_DEC_RET_SUCCESS, VPU_DEC_RET_FAILURE…

2.3 Encoder API Functions

2.3.1 Encoder Open and Close

2.3.1.1 VPU_EncGetVersionInfo

```
VpuEncRetCode VPU_EncGetVersionInfo (
  VpuVersionInfo     * pOutVerInfo)
```

Description:

Function to get the vpulib and firmware version.

Arguments

- pOutVerInfo [out] - Pointer to VpuVersionInfo struct where the output parameters will be stored.

Return value

VPU_ENC_RET_SUCCESS

2.3.1.2 VPU_EncGetWrapperVersionInfo

```
VpuEncRetCode VPU_EncGetWrapperVersionInfo (
  VpuWrapperVersionInfo   * pOutVerInfo)
```

Description:

Function to get the VPU wrapper version.

Arguments

- pOutVerInfo [out] - Pointer to VpuWrapperVersionInfo struct where the output parameters will be stored.

Return value

VPU_ENC_RET_SUCCESS

2.3.1.3 VPU_EncGetInitialInfo

```
VpuEncRetCode VPU_EncGetInitialInfo (
  VpuEncHandle     InHandle,
  VpuEncInitInfo     * pOutInitInfo)
```

Description:

Function to get the initial information.

Arguments:

- InHandle [in] - Handle of VPU encoder.
- pOutInitInfo [out] - Pointer to VpuEncInitInfo struct where initial configuration parameters of the encoder will be stored.

Return value:

VPU_ENC_RET_SUCCESS

2.3.1.4 VPU_EncConfig

```
VpuEncRetCode VPU_EncConfig (
  VpuEncHandle     InHandle,
  VpuEncConfig     InEncConf,
  Void      * pInParam)
```

Description:

Function to set the corresponding parameters of encoder according to InEncConf type.

Arguments:

- InEncConf [in] - Type of configuration to be set.
- pInParam [in] - Value that used to set the encoder according to InEncConf type.
- InHandle [out] - Handle of VPU encoder.

Return value:

VpuEncRetCode - value: VPU_ENC_RET_SUCCESS, VPU_ENC_RET_FAILURE…

2.3.1.5 VPU_EncOpen

```
VpuncRetCode VPU_EncOpen (
  VpuEncHandle     * pOutHandle,
  VpuMemInfo    * pInMemInfo,
  VpuEncOpenParam  * pInParam)
```

Description:

Function to open new VPU handle, called by function VPU_EncOpenSimp.

Arguments:

- pInMemInfo [in] - Pointer to VpuMemInfo struct which contains the memory information.
- pInParam [in] - Pointer to VpuEncOpenParam struct which contains input parameters of the encoder.
- pOutHandle [out] - Handle of VPU encoder.

Return value:

VpuEncRetCode - value: VPU_ENC_RET_SUCCESS, VPU_ENC_RET_FAILURE…

2.3.1.6 VPU_EncOpenSimp

```
VpuEncRetCode VPU_EncOpenSimp (
  VpuEncHandle       * pOutHandle,
  VpuMemInfo      * pInMemInfo,
  VpuEncOpenParamSimp   * pInParam)
```

Description:

Function to open new VPU handle which is realized in function VPU_EncOpen.

Arguments:

- pInMemInfo [in] - Pointer to VpuMemInfo struct which contains the memory information.
- pInParam [in] - Pointer to VpuEncOpenParam struct which contains input parameters of the encoder.
- pOutHandle [out] - Handle of VPU encoder.

Return value:

VpuEncRetCode - value: VPU_ENC_RET_SUCCESS, VPU_ENC_RET_FAILURE…

2.3.1.7 VPU_EncLoad

```
VpuEncRetCode VPU_EncLoad ()
```

Description:

Function to parse the log level, load VPU.

Arguments:

None.

Return value:

VPU_ENC_RET_SUCCESS

2.3.1.8 VPU_EncUnLoad

```
VpuEncRetCode VPU_EncUnLoad ()
```

Description:

Function to unload VPU.

Arguments:

None.

Return value:

VPU_ENC_RET_SUCCESS

2.3.1.9 VPU_EncReset

```
VpuEncRetCode VPU_EncReset (
  VpuEncHandle     InHandle)
```

Description:

Function to reset the handle of VPU. H1 encoder has no interface to reset.

Arguments:

- InHandle [in] - Handle of VPU encoder.

Return value:

VPU_ENC_RET_SUCCESS

2.3.1.10 VPU_EncClose

```
VpuEncRetCode VPU_EncClose (
  VpuEncHandle     InHandle)
```

Description:

Function to close the encoder.

Arguments:

- InHandle [in] - Handle of VPU encoder.

Return value:

VPU_ENC_RET_SUCCESS

2.3.2 Encode

2.3.2.1 VPU_EncEncodeFrame

```
VpuEncRetCode VPU_EncEncodeFrame (
  VpuEncHandle     InHandle,
  VpuEncEncParam  * pInOutParam)
```

Description:

Function to encode one frame. Before encoding the first frame, call function VPU_EncStartEncode to start stream. Otherwise, call function VPU_EncDoEncode.

Arguments:

- InHandle [in/out] - Handle of the VPU encoder.
- pInOutParam [out] - Pointer to VpuEncEncParam struct where input and output parameters are stored.

Return value:

VpuEncRetCode - value: VPU_ENC_RET_SUCCESS, VPU_ENC_RET_FAILURE…

2.3.3 Memory Query and Free

2.3.3.1 VPU_EncQueryMem

```
VpuEncRetCode VPU_EncQueryMem (
  VpuMemInfo    * pOutMemInfo)
```

Description:

Function to query output physical and virtual memory information to further allocate.

Arguments:

- pOutMemInfo [out] - Pointer to VpuMemInfo struct where the output memory information is stored.

Return value:

VPU_ENC_RET_SUCCESS

2.3.3.2 VPU_EncFreeMem

```
VpuEncRetCode VPU_EncFreeMem (
  VpuMemDesc    * pInMem)
```

Description:

Function to free memory.

Arguments:

- pInMem [in] - Pointer to VpuMemDesc struct where the memory information is stored.

Return value:

VpuEncRetCode - value: VPU_ENC_RET_SUCCESS, VPU_ENC_RET_FAILURE…

2.3.3.3 VPU_EncRegisterFrameBuffer

```
VpuEncRetCode VPU_EncRegisterFrameBuffer (
  VpuEncHandle     InHandle,
  VpuFrameBuffer   * pInFrameBufArray,
  int       nNum,
  int       nSrcStride)
```

Description:

Function to register frame buffer.

Arguments:

- InHandle [in/out] - Handle of VPU encoder.
- pInFrameBufArray [in] - Pointer to VpuFrameBuffer struct where the information of VPU frame buffer is stored.
- nNum [in] - Number of minimum frame buffer count obtained from initial information.
- nSrcStride [in] - Number of stride of plane 0.

Return value:

VPU_ENC_RET_SUCCESS

2.4 API Calling Sequence

This section gives the API calling sequence to further understand encoding and decoding process

2.4.1 Decoding Calling Sequence

The following figure shows the block diagram of a typical decoding process.

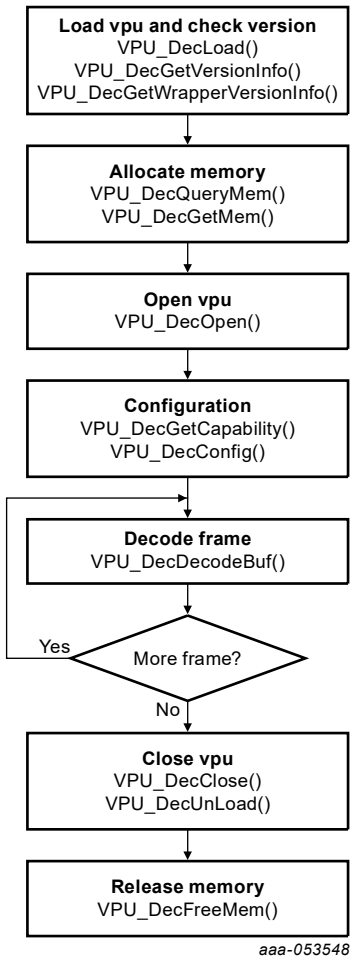

Figure 2. Typical decoding process

The main steps of decoding are as follows:

1. Load the VPU.
   VPU_DecLoad()

2. Check the vpulib and VPU wrapper version.
   VPU_DecGetVersionInfo()
   VPU_DecGetWrapperVersionInfo()
3. Query memory to hold for output buffer.
   VPU_DecQueryMem()
   Note: *Query the memory information from VPU with VPU_EncQueryMem(), and then the VPU will allocate internal memory for it.*
4. Allocate memory for the VPU.
   VPU_DecGetMem()
5. Open new VPU decoder handle.
   VPU_DecOpen()
   Note: *Configuration parameters will be set according to video type and decode handle will be created.*
6. Set decoding configuration for the VPU decoder.
   VPU_DecGetCapability()
   VPU_DecConfig()
7. Start to decode the frame.
   VPU_DecDecodeBuf()
   Note: *Function VPU_DecDecodeBuf() includes the following steps:*
   a. *Scan whether there is decoded frame.If there is decoded frame, the output state will have the state VPU_DEC_OUTPUT_DIS to indicate there is a frame to be sent out.*
   b. *Process the current frame input data. The output state will have the state VPU_DEC_INPUT_USED.*
   c. *Decode the current frame. If the input data is consumed and the frame is decoded successfully, the output state will have the state VPU_DEC_ONE_FRM_CONSUMED.*
8. After decoding, perform different actions according to the output state.
   VPU_DecGetOutputFrame()
   VPU_DecGetConsumedFrameInfo()
   VPU_DecFlushAll()
   Note: *The following are several actions that can happen with the output state pOutBufRetCode.*
   - *If (pOutBufRetCode & VPU_DEC_ONE_FRM_CONSUMED) is not 0, then call VPU_DecGetConsumedFrameInfo() to get the frame information in VPU.*
   - *If (pOutBufRetCode & VPU_DEC_OUTPUT_DIS) is not 0, then call VPU_DecGetOutputFrame() to get the output frame information.*
   - *If (pOutBufRetCode & VPU_DEC_FLUSH) is not 0, then call VPU_DecFlushAll() to flush the decoder.*
9. Release the frame buffer to the VPU after displaying the frame.
   VPU_DecOutFrameDisplayed()
10. Close the VPU.
    VPU_DecClose()
    VPU_DecUnLoad()
11. Release the memory.
    VPU_DecFreeMem()

2.4.2 Encoding Calling Sequence

The following figure shows the block diagram of a typical encoding process.

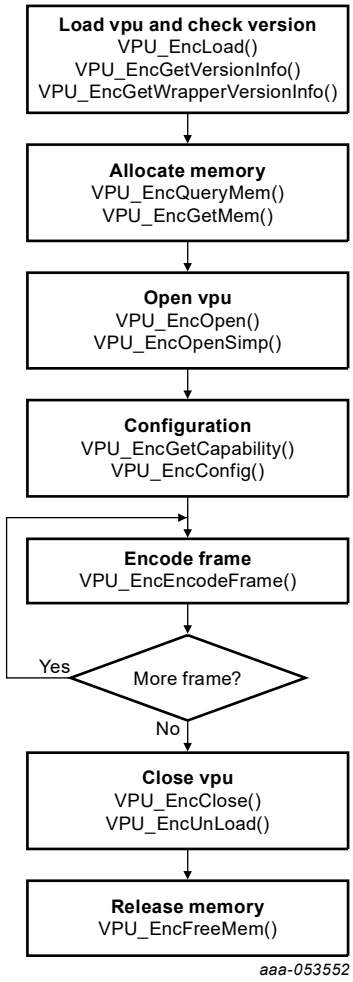

Figure 3. Typical encoding process

The main steps of encoding are as follows:

1. Load the VPU.
   VPU_EncLoad()
2. Check the vpulib and VPU wrapper version information.
   VPU_EncGetVersionInfo()
   PU_EncGetWrapperVersionInfo()
3. Query the memory to hold for the output buffer.
   VPU_EncQueryMem()
   Note: *Query the memory information from the VPU with VPU_EncQueryMem(), and then the VPU will allocate the internal memory for it.*
4. Allocate memory for the VPU.
   VPU_EncGetMem()
5. Open the new VPU encoder handle.
   VPU_EncOpen()
   VPU_EncOpenSimp()
   Note: *Configuration parameters will be set according to the encoded stream type and encode handle will be created.*
6. Set the default configuration and get the initial information of the VPU encoder.
   VPU_EncConfig()

VPU_EncGetInitialInfo()

7. Allocate physical memory for the input buffer and register the frame buffer if needed.
   VPU_EncRegisterFrameBuffer()
   Note: *H1 encoder does not need to register the frame buffer, so VPU_EncRegisterFrameBuffer() does nothing.*
8. Start to encode stream.
   VPU_EncEncodeFrame()
   Note: *This is the stream production process, including stream start and stream encode. Encoder first starts stream and outputs codec data, and then starts to encode stream one frame by one frame.*
9. Close the VPU.
   VPU_EncClose()
   VPU_EncUnLoad()
10. Release the VPU memory.
    PU_EncFreeMem()

3 Amphion VPU Interface

The Amphion VPU is supported on the i.MX 8QuadMax and 8QuadXPlus platforms using the Malone hardware for decoding and Windsor hardware for encoding. There are dedicated Cortex-M cores reserved for Amphion VPU for decoding and encoding.

Both hardware decoder and encoder are controlled by firmware running on the Cortex-M cores and all APIs are exported to Arm cores through the RPC protocol, which is implemented through shared memory and MU interrupt. All RPC communication protocol are defined through some configuration parameters, commands, and callback event (e.g., message), etc. Applications on Arm cores send commands to decoder on Cortex-M cores, such as start, stop, etc. and decoder will send callback events to response application, such as start done, request frame buffers, frame ready, etc. All input and output parameters will be transferred through RPC shared memory.

Decoder and encoder state machines should be maintained through one event handler implemented on Arm cores.

3.1 Amphion RPC Protocol

For the RPC protocol, there is one header file ‘mediasys_types.h’ used to define commands, events and all required structures stored in shared memory.

3.1.1 RPC Shared Memory Interface

One main structure is used to build the index for all data, including ring buffer descriptor. All shared information between the application and Cortex-M cores can be fetched from the interface structure. Since there are no MMU on the Cortex-M cores, all memory should be physical continuous, including interface struct and all ring buffers.

The following is the shared memory space for the RPC decoder interface.

```
typedef struct {
  u_int32 FwExecBaseAddr;
  u_int32 FwExecAreaSize;
  MediaIPFW_Video_BufDesc StreamCmdBufferDesc;
  MediaIPFW_Video_BufDesc StreamMsgBufferDesc;
  u_int32 StreamCmdIntEnable[VID_API_NUM_STREAMS];
  MediaIPFW_Video_PitchInfo StreamPitchInfo[VID_API_NUM_STREAMS];
```

```
    u_int32 StreamConfig[VID_API_NUM_STREAMS];
    MediaIPFW_Video_CodecParamTabDesc CodecParamTabDesc;
    MediaIPFW_Video_JpegParamTabDesc JpegParamTabDesc;
    u_int32 pStreamBuffDesc[VID_API_NUM_STREAMS][VID_API_MAX_BUF_PER_STR];
    MediaIPFW_Video_SeqInfoBuffTabDesc SeqInfoTabDesc;
    MediaIPFW_Video_PicInfoBuffTabDesc PicInfoTabDesc;
    MediaIPFW_Video_GopInfoBuffTabDesc GopInfoTabDesc;
    MediaIPFW_Video_QMeterInfoTabDesc QMeterInfoTabDesc;
    u_int32 StreamError[VID_API_NUM_STREAMS];
    u_int32 FWVersion;
    u_int32 uMVDMipsOffset;
    u_int32 uMaxDecoderStreams;
    MediaIPFW_Video_DbgLogDesc DbgLogDesc;
    MediaIPFW_Video_FrameBuffer StreamFrameBuffer[VID_API_NUM_STREAMS];
    MediaIPFW_Video_FrameBuffer StreamDCPBuffer[VID_API_NUM_STREAMS];
    MediaIPFW_Video_UData UDataBuffer[VID_API_NUM_STREAMS];
    MediaIPFW_Video_BufDesc DebugBufferDesc;
    MediaIPFW_Video_BufDesc EngAccessBufferDesc[VID_API_NUM_STREAMS];
    u_int32 ptEncryptInfo[VID_API_NUM_STREAMS];
    MEDIAIP_FW_SYSTE-M_CONFIG sSystemCfg;
    u_int32 uApiVers-ion;
    BUFFER_INFO_TYPE StreamBuffInfo[VID_API_NUM_STREAMS];
  } DEC_RPC_HOST_IFACE, *pDEC_RPC_HOST_IFACE;
```

The following is the shared memory space for the RPC Encoder interface.

```
  typedef struct {
    u_int32          FwExecBaseAddr;
    u_int32          FwExecAreaSize;
    BUFFER_DESCRIPTOR_TYPE      StreamCmdBufferDesc;
    BUFFER_DESCRIPTOR_TYPE      StreamMsgBufferDesc;
    u_int32          StreamCmdIntEnable[VID_API_NUM_STREAMS];
    u_int32          FWVersion;
    u_int32          uMVDFWOffset;
    u_int32          uMaxEncoderStreams;
    u_int32          pEncCtrlInterface[VID_API_NUM_STREAMS];
    MEDIAIP_FW_SYSTEM_CONFIG    sSystemCfg;
    u_int32          uApiVersion;
    BUFFER_DESCRIPTOR_TYPE      DebugBufferDesc;
  } ENC_RPC_HOST_IFACE, *pENC_RPC_HOST_IFACE;
```

3.1.1.1 Sample Code for RPC Decoder Interface Initialization

The following is sample code to initialize the RPC interface with continuous physical memory: Application is responsible to record corresponding virtual address for RPC communication.

```
  pSharedInterface = (pDEC_RPC_HOST_IFACE)rpc_virt_addr;
  base_phy_addr = rpc_phy_addr - fw_binary_phy_addr;
  pSharedInterface->FwExecBaseAddr = base_phy_addr;
  pSharedInterface->FwExecAreaSize = total_size;
  pSharedCmdBufDescPtr = (MediaIPFW_Video_BufDesc *)&pSharedInterface-
>StreamCmdBufferDesc;
  pSharedMsgBufDescPtr = (MediaIPFW_Video_BufDesc *)&pSharedInterface-
>StreamMsgBufferDesc;
  phy_addr = base_phy_addr + sizeof(DEC_RPC_HOST_IFACE);
  pSharedCmdBufDescPtr->uWrPtr = phy_addr;
```

```
  pSharedCmdBufDescPtr->uRdPtr = pSharedCmdBufDescPtr->uWrPtr;
  pSharedCmdBufDescPtr->uStart = pSharedCmdBufDescPtr->uWrPtr;
  pSharedCmdBufDescPtr->uEnd = pSharedCmdBufDescPtr->uStart + CMD_SIZE;
  phy_addr += CMD_SIZE;
  pSharedMsgBufDescPtr->uWrPtr = phy_addr;
  pSharedMsgBufDescPtr->uRdPtr = pSharedMsgBufDescPtr->uWrPtr;
  pSharedMsgBufDescPtr->uStart = pSharedMsgBufDescPtr->uWrPtr;
  pSharedMsgBufDescPtr->uEnd = pSharedMsgBufDescPtr->uStart + MSG_SIZE;
  phy_addr += MSG_SIZE;
  pSharedInterface->CodecParamTabDesc.pCodecParamArrayBase = phy_addr;
  phy_addr += CODEC_SIZE;
  pSharedInterface->JpegParamTabDesc.pJpegParamArrayBase = phy_addr;
  phy_addr += JPEG_SIZE;
  pSharedInterface->SeqInfoTabDesc.pSeqInfoArrayBase = phy_addr;
  phy_addr += SEQ_SIZE;
  pSharedInterface->PicInfoTabDesc.pPicInfoArrayBase = phy_addr;
  phy_addr += PIC_SIZE;
  pSharedInterface->GopInfoTabDesc.pGopInfoArrayBase = phy_addr;
  phy_addr += GOP_SIZE;
  pSharedInterface->QMeterInfoTabDesc.pQMeterInfoArrayBase = phy_addr;
  phy_addr += QMETER_SIZE;
  pSharedInterface->DbgLogDesc.uDecStatusLogBase = phy_addr; //set NULL to
 disable
  pSharedInterface->DbgLogDesc.uDecStatusLogSize = DBGLOG_SIZE; //set 0 to
 disable
  pSharedInterface->uDecStatusLogLevel = 0;
  phy_addr += DBGLOG_SIZE;
  pDebugBufferDesc = &pSharedInterface->DebugBufferDesc;
  pDebugBufferDesc->uWrPtr = phy_addr;
  pDebugBufferDesc->uRdPtr = pDebugBufferDesc->uWrPtr;
  pDebugBufferDesc->uStart = pDebugBufferDesc->uWrPtr; //set NULL to disable
  pDebugBufferDesc->uEnd = pDebugBufferDesc->uStart + DEBUG_SIZE;
  phy_addr += DEBUG_SIZE;
  for (i = 0; i < VPU_MAX_NUM_STREAMS; i++) {
    pEngAccessBufferDesc = &pSharedInterface->EngAccessBufferDesc[i];
    pEngAccessBufferDesc->uWrPtr = phy_addr;
    pEngAccessBufferDesc->uRdPtr = pEngAccessBufferDesc->uWrPtr;
    pEngAccessBufferDesc->uStart = pEngAccessBufferDesc->uWrPtr;
    pEngAccessBufferDesc->uEnd = pEngAccessBufferDesc->uStart + ENG_SIZE;
    phy_addr += ENG_SIZE;
  }
  for (i = 0; i < VPU_MAX_NUM_STREAMS; i++) {
    pSharedInterface->ptEncryptInfo[i] = phy_addr;
    phy_addr += sizeof(MediaIPFW_Video_Encrypt_Info);
  }
```

3.1.1.2 Sample Code for Decoder System Configuration Parameter Initialization

The following is the sample code to initialize the decoder system configuration parameter.

```
  pSharedInterface = (pDEC_RPC_HOST_IFACE)rpc_virt_addr;
  regs_base = 0x40000000;
  pSystemCfg = &pSharedInterface->sSystemCfg;
  pSystemCfg->uNumMalones = 1;
  pSystemCfg->uMaloneBaseAddress[0] = (unsigned int)(regs_base + 0x180000);
  pSystemCfg->uMaloneBaseAddress[0x1] = 0x0;
  pSystemCfg->uHifOffset[0x0] = 0x1C000;
  pSystemCfg->uHifOffset[0x1] = 0x0;
```

```
  pSystemCfg->uDPVBaseAddr = 0x0;
  pSystemCfg->uDPVIrqPin = 0x0;
  pSystemCfg->uPixIfBaseAddr = (unsigned int)(regs_base + 0x180000 + 0x20000);
  pSystemCfg->uFSLCacheBaseAddr[0] = (unsigned int)(regs_base + 0x60000);
  pSystemCfg->uFSLCacheBaseAddr[1] = (unsigned int)(regs_base + 0x68000);
```

3.1.1.3 Sample Code for RPC Encoder Interface Initialization

Application is responsible to record corresponding virtual address for RPC communication.

```
  pSharedInterface = (pENC_RPC_HOST_IFACE)rpc_virt_addr;
  base_phy_addr = rpc_phy_addr - fw_binary_phy_addr;
  pSharedInterface->FwExecBaseAddr = base_phy_addr;
  pSharedInterface->FwExecAreaSize = total_size;
  pSharedCmdBufDescPtr = (BUFFER_DESCRIPTOR_TYPE *)&pSharedInterface-
>StreamCmdBufferDesc;
  pSharedMsgBufDescPtr = (BUFFER_DESCRIPTOR_TYPE *)&pSharedInterface-
>StreamMsgBufferDesc;
  phy_addr = base_phy_addr + sizeof(ENC_RPC_HOST_IFACE);
  pSharedCmdBufDescPtr->wptr = phy_addr;
  pSharedCmdBufDescPtr->rptr = pSharedCmdBufDescPtr->wptr;
  pSharedCmdBufDescPtr->start = pSharedCmdBufDescPtr->wptr;
  pSharedCmdBufDescPtr->end = pSharedCmdBufDescPtr->start + CMD_SIZE;
  phy_addr += CMD_SIZE;
  pSharedMsgBufDescPtr->wptr = phy_addr;
  pSharedMsgBufDescPtr->rptr = pSharedMsgBufDescPtr->wptr;
  pSharedMsgBufDescPtr->start = pSharedMsgBufDescPtr->wptr;
  pSharedMsgBufDescPtr->end = pSharedMsgBufDescPtr->start + MSG_SIZE;
  phy_addr += MSG_SIZE;
  for (i = 0; i < VID_API_NUM_STREAMS; i++) {
    pSharedInterface->pEncCtrlInterface[i] = phy_addr;
    phy_addr += sizeof(MEDIA_ENC_API_CONTROL_INTERFACE);
  }
  for (i = 0; i < VID_API_NUM_STREAMS; i++) {
    temp_addr = pSharedInterface->pEncCtrlInterface[i];
    pEncCtrlInterface = (pMEDIA_ENC_API_CONTROL_INTERFACE)
(phy_to_virt(temp_addr));
    pEncCtrlInterface->pEncYUVBufferDesc = phy_addr;
    phy_addr += sizeof(MEDIAIP_ENC_YUV_BUFFER_DESC);
    pEncCtrlInterface->pEncStreamBufferDesc = phy_addr;
    phy_addr += sizeof(BUFFER_DESCRIPTOR_TYPE);
    pEncCtrlInterface->pEncExpertModeParam = phy_addr;
    phy_addr += sizeof(MEDIAIP_ENC_EXPERT_MODE_PARAM);
    pEncCtrlInterface->pEncParam = phy_addr;
    phy_addr += sizeof(MEDIAIP_ENC_PARAM);
    pEncCtrlInterface->pEncMemPool = phy_addr;
    phy_addr += sizeof(MEDIAIP_ENC_MEM_POOL);
    pEncCtrlInterface->pEncEncodingStatus = phy_addr;
    phy_addr += sizeof(ENC_ENCODING_STATUS);
    pEncCtrlInterface->pEncDSAStatus = phy_addr;
    phy_addr += sizeof(ENC_DSA_STATUS_t);
  }
```

3.1.1.4 Sample Code for Encoder System Configuration Parameter Initialization

The following is sample code for initializing the encoder configuration parameter for Windsor.

```
MEDIAIP_FW_SYSTEM_CONFIG *pSystemCfg;
regs_base = 0x40000000;
  pSharedInterface = (pENC_RPC_HOST_IFACE)rpc_virt_addr;
  pSystemCfg = &pSharedInterface->sSystemCfg;
  pSystemCfg->uNumWindsors = 1;
  pSystemCfg->uWindsorIrqPin[0x0][0x0] = 0x4; // PAL_IRQ_WINDSOR_LOW
  pSystemCfg->uWindsorIrqPin[0x0][0x1] = 0x5; // PAL_IRQ_WINDSOR_HI
  if (encoder_id == 0)
    pSystemCfg->uWindsorBaseAddress[0] = (unsigned int)(regs_base + 0x800000);
  else
    pSystemCfg->uWindsorBaseAddress[0] = (unsigned int)(regs_base + 0xa00000);
```

There is one encoder engine on i.MX 8QuadXPlus and two encoder engines on i.MX 8QuadMax.

3.1.2 RPC Commands

RPC commands and events are the basic communication between the application and Cortex-M cores. Applications send commands to the Cortex-M cores and receive events from the Cortex-M cores using the enumerations as shown below.

The following are the command types for decoder and encoder RPC commands.

```
  typedef enum {
    VID_API_CMD_XXX = xxx,
  } TB_API_DEC_CMD;
  typedef enum {
    VID_API_EVENT_XXX = xxx,
  } TB_API_DEC_EVENT;
  typedef enum {
    GTB_ENC_CMD_XXX  =  xxx,
  } GTB_ENC_CMD;
  typedef enum {
    VID_API_ENC_EVENT_XXX = xxx,
  } ENC_TB_API_ENC_EVENT;
```

3.1.3 RPC MU

MU modules are designed to trigger interrupt between different processors on i.MX 8 platforms. In the i.MX RPC implementation, MU interrupt (RX0) will be used to replace CPU polling. The application is responsible to register interrupt handler, which is used to receive RPC message and trigger the state machine of decoder (event handler).

MU configuration:

```
  QXP:
    MU0: irq: (469+32); base: 0x2d000000 //for M core ID 0
    MU1: irq: (470+32); base: 0x2d020000 //for M core ID 1
  QM:
    MU0: irq: (472+32); base: 0x2d000000
    MU1: irq: (473+32); base: 0x2d020000
    MU2: irq: (474+32); base: 0x2d040000
```

```
DM:
  MU0: irq: (472+32); base: 0x2d000000
  MU1: irq: (473+32); base: 0x2d020000
```

For the VPU decoder on i.MX 8QuadXPlus, to use the Cortex-M core ID 0, set the MU interrupt IRQ to 501, and mu_phy_base to 0x2d000000. Only regindex '0' is enabled to trigger interrupt in RPC communication. For encoder on i.MX 8QuadXPlus, to use the Cortex-M core ID 1, set MU interrupt IRQ to 502, and mu_phy_base to 0x2d020000.

Sample code to initialize the MU module:

```
MU_Init(mu_virt_base);
MU_EnableRxFullInt(mu_virt_base, 0); //only RX0 enabled
Sample code to send/receive message:
MU_SendMessage(mu_virt_base, regIndex, msg);
MU_ReceiveMsg(mu_virt_base, 0, &msg)
```

Send msg definition:

```
Regindex:
  0: used to transfer msg
  1: used to transfer paramter
Msg:
  1: init done, and no parameter is required in regindex 1
  2: set the RPC buffer offset, and the parameter is the RPC physical address
from M0+ sight
  3: set the boot address, and the parameter is the binary load physcial
address from Arm cores sight
  4: command, notify the Cortex-M core to read the RPC command buffer, and no
parameter is required in regindex 1
```

Receive msg definition:

```
0xAA: M core boot done, waiting the rpc config
0x55: M core start done, waiting decoder command
0xA5: M core snapshot done, for suspend
0x5: M core send event, Arm core need to read rpc message buffer
```

For more information about how to send/receive MU interrupt, see the i.MX *Linux Reference Manual* (IMXLXRM).

3.1.4 RPC Message

Command and event are transferred through RPC command ring buffer and message ring buffer separately.

Sample code to send command (followed by MU sending msg: 4):

```
MediaIPFW_Video_BufDesc *pCmdDesc = &pSharedInterface->StreamCmdBufferDesc;
u_int32 *cmddata;
u_int32 i;
u_int32 *cmdword=(u_int32 *)(cmd_mem_vir+pCmdDesc->uWrPtr - pCmdDesc->uStart);
*cmdword = 0;
*cmdword |= ((strIdx & 0x000000ff) << 24); //stream index
*cmdword |= ((cmdnum & 0x000000ff) << 16);
```

```
  *cmdword |= ((cmdid & 0x00003fff) <<  0); // TB_API_DEC_CMD
  pCmdDesc->uWrPtr = (pCmdDesc->uWrPtr + 4)%(ring_buffer_size);
  for (i = 0; i < cmdnum; i++) {
    cmddata = (u_int32 *)(cmd_mem_vir+pCmdDesc->uWrPtr - pCmdDesc->uStart);
    *cmddata = local_cmddata[i];
    pCmdDesc->uWrPtr = (pCmdDesc->uWrPtr + 4)%(ring_buffer_size);
  }
```

Sample code to receive message (followed by MU receiving msg: 0x5):

```
  MediaIPFW_Video_BufDesc *pMsgDesc = &pSharedInterface->StreamMsgBufferDesc;
  u_int32 msgword =*((u_int32 *)(msg_mem_vir+pMsgDesc->uRdPtr-pMsgDesc-
>uStart));
  msg->idx = ((msgword & 0xff000000) << 24); //stream index
  msg->msgnum = ((msgword & 0x00ff0000) << 16);
  msg->msgid = ((msgword & 0x00003fff) << 0); // TB_API_DEC_EVENT
  pMsgDesc->uRdPtr = (pMsgDesc -> uRdPtr + 4)%(ring_buffer_size);
  for (i = 0; i < msg->msgnum; i++) {
    msg->msgdata[i]=*((u_int32*)(msg_mem_vir+pMsgDesc->uRdPtr-pMsgDesc-
>uStart));
  pMsgDesc->uRdPtr = (pMsgDesc->uRdPtr + 4)%(ring_buffer_size);
  } msg->msgid = ((msgword & 0x00003fff) >> 0); // TB_API_DEC_EVENT
  pMsgDesc->uRdPtr = (pMsgDesc -> uRdPtr + 4)%(ring_buffer_size);
  for (i = 0; i < msg->msgnum; i++) {
    msg->msgdata[i]=*((u_int32*)(msg_mem_vir+pMsgDesc->uRdPtr-pMsgDesc-
>uStart));
    pMsgDesc->uRdPtr = (pMsgDesc -> uRdPtr + 4)%(ring_buffer_size);
  }
```

3.2 Cortex-M Core Boot

Some internal protocols are defined to make the Cortex-M core boot properly. Applications should perform the steps below to follow the protocol.

1. Application should load firmware binary into one physical continuous memory firstly and then set the physical address to offset register of the Cortex-M core, finally boot the Cortex-M core through clear wait register of the Cortex-M core.

```
CSR config:
QXP:
CSR0: 0x2d040000
CSR1: 0x2d050000
QM:
CSR0: 0x2d080000
CSR1: 0x2d090000
CSR2: 0x2d0a0000
DM:
CSR0: 0x2d040000
CSR1: 0x2d050000
Sample code to use the Cortex-M core ID 0 on QXP:
csr_base = 0x2d040000;
csr_wait = 0x2d040004;
*(u_int32*)csr_base = fw_binary_phy_addr;
*(u_int32*)csr_wait = 0;
```

2. Application must wait until receiving special message (0xAA). This means the Cortex-M core is already initialized. Then application can initialize RPC interface from previous section.

3. Application sends firmware binary physical address and RPC physical address to the Cortex-M core through MU_SendMessage.

```
Sample code:
MU_SendMessage(mu_virt_base, 1, fw_binary_phy_addr);
MU_SendMessage(mu_virt_base, 0, 3);
MU_SendMessage(mu_virt_base, 1, rpc_phy_addr);
MU_SendMessage(mu_virt_base, 0, 2);
MU_SendMessage(mu_virt_base, 1, 2);
MU_SendMessage(mu_virt_base, 0, 1);
```

4. Application waits until receiving special message (0x55) from the Cortex-M core, which indicates that the Cortex-M core is already started up. After these steps are complete, bootup of the Cortex-M core is finished, and the application can send normal decoder command.

3.3 Decoder Workflow

3.3.1 Stream Config

Application sets initial parameter for every stream instance before start. Below is Sample code to initialize the decoder:

```
  u_int32 *CurrStrfg;
  pSharedInterface = (pDEC_RPC_HOST_IFACE)Interface;
  CurrStrfg = &pSharedInterface->StreamConfig[str_idx];
  *CurrStrfg = 0;
  VID_STREAM_CONFIG_STRBUFIDX_SET(0, CurrStrfg);
  VID_STREAM_CONFIG_NOSEQ_SET(FALSE, CurrStrfg);
  VID_STREAM_CONFIG_DEBLOCK_SET(FALSE, CurrStrfg);
  VID_STREAM_CONFIG_DERING_SET(FALSE, CurrStrfg);
  VID_STREAM_CONFIG_PLAY_MODE_SET(MEDIA_PLAYER_API_MODE_CONTINUOUS,CurrStrfg);
  VID_STREAM_CONFIG_FS_CTRL_MODE_SET(MEDIA_PLAYER_FS_CTRL_MODE_EXTERNAL,
CurrStrfg);
  VID_STREAM_CONFIG_ENABLE_DCP_SET(TRUE, CurrStrfg);
  VID_STREAM_CONFIG_NUM_STR_BUF_SET(1, CurrStrfg);
  VID_STREAM_CONFIG_MALONE_USAGE_SET(1, CurrStrfg);
  VID_STREAM_CONFIG_MULTI_VID_SET(FALSE, CurrStrfg);
  VID_STREAM_CONFIG_OBFUSC_EN_SET(FALSE, CurrStrfg);
  VID_STREAM_CONFIG_RC4_EN_SET(FALSE, CurrStrfg);
  VID_STREAM_CONFIG_MCX_SET(TRUE, CurrStrfg);
  VID_STREAM_CONFIG_PES_SET(FALSE, CurrStrfg);
  VID_STREAM_CONFIG_NUM_DBE_SET(1, CurrStrfg);
  VID_STREAM_CONFIG_FORMAT_SET(VSys_AvcFrmt, CurrStrfg); //for h.264 format
```

3.3.2 Event Handler

For normal decoder workflow, the application should implement one event handler to respond the event from the Cortex-M core and send proper command to trigger correct decoder state. The following figure shows one simple flow for normal decoder playback.

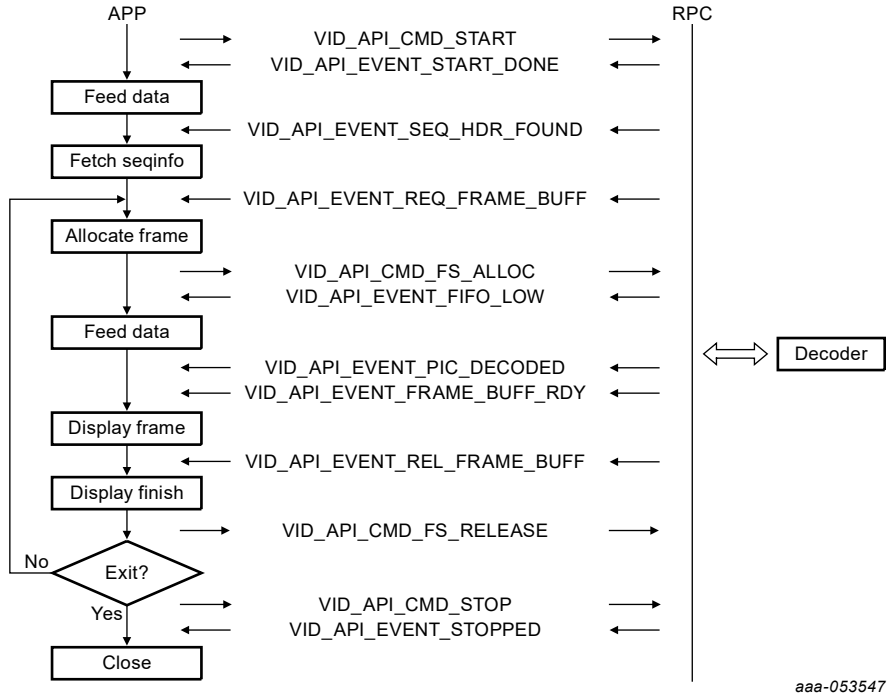

Figure 4. Simple flow for normal decoder playback

For dynamic resolution changes, event ‘VID_API_EVENT_RES_CHANGE’ will be sent and thus allocation of memory needs to change.

To support other advanced features, such as flush operation, command ‘VID_API_CMD_ABORT’ should be used and decoder will send callback event VID_API_EVENT_STR_BUF_RST when the command is processed.

Input data buffer:

All input data are maintained by ring buffers, which are managed by read and write pointer. The buffer descriptors are defined in pSharedInterface->pStreamBuffDesc[]. Every descriptor needs to point to one HW MCX register: (total 8 groups of MCX registers)

```
regs_base + DEC_MFD_XREG_SLV_BASE + MFD_MCX + MFD_MCX_OFF * strIdx
  = 0x2c000000 + 0x00180000 + 0x00020800 + 0x20 * strIdx
```

Output frame buffer:

For internal mode (obsolete), one large memory pool is registered through pSharedInterface->StreamFrame Buffer[], and decoder will divide frames from the large memory pool internally. For external mode, frames are registered through command ‘VID_API_CMD_FS_ALLOC’, and application is responsible to allocate frame one by one following the callback event request.

3.3.2.1 VID_API_EVENT_REQ_FRAME_BUFF

Decoder requests frame and applications sends buffer through command ‘VID_API_CMD_FS_ALLOC’. There are three types of buffers:

- DCP

```
uLocalCmdBuffer[0] = Id;
```

```
uLocalCmdBuffer[1] = dcp_phy;
uLocalCmdBuffer[2] = dcp_size;
uLocalCmdBuffer[3] = 0;
uLocalCmdBuffer[4] = 0;
uLocalCmdBuffer[5] = 0;
uLocalCmdBuffer[6] = ulFsType;  // MEDIAIP_MEM_REQ : 0: Frame; 1: MBI; 2: DCP
```

DCP buffer is only requested by HEVC decoder, and DCP size is one constant value ‘0x3000000’ for the worst case. And the maximum DCP buffer number should be less than 3.

- MBI

```
uLocalCmdBuffer[0] = Id;
uLocalCmdBuffer[1] = mbi_phy;
uLocalCmdBuffer[2] = mbi_size;
uLocalCmdBuffer[3] = 0;
uLocalCmdBuffer[4] = 0;
uLocalCmdBuffer[5] = 0;
uLocalCmdBuffer[6] = ulFsType;  // MEDIAIP_MEM_REQ : 0: Frame; 1: MBI; 2: DCP
the maximum MBI buffer number should be less than 19.
```

- Frame:

```
uLocalCmdBuffer[0] = ulFsId;
uLocalCmdBuffer[1] = ulFsLumaBase[0];
uLocalCmdBuffer[2] = ulFsLumaBase[1];  //for field
uLocalCmdBuffer[3] = ulFsChromaBase[0];
uLocalCmdBuffer[4] = ulFsChromaBase[1];  //for field
uLocalCmdBuffer[5] = ulFsStride;
uLocalCmdBuffer[6] = ulFsType;  // MEDIAIP_MEM_REQ : 0: Frame; 1: MBI; 2: DCP
```

Sample code to calculate frame buffer stride/size and MBI size:

```
 u_int32 uVertAlign = 256-1;
 u_int32 uMBIAlign = 0x800-1;
 bool b10BitFormat = (pSeqinfo->uBitDepthLuma > 8) || pSeqinfo->uBitDepthChroma
>8);
 width = b10BitFormat ? (width + ((width + 3) >> 2)): width;
 width = ((width + uVertAlign) & ~uVertAlign);
 stride = width;
 height = ((height + uVertAlign) & ~uVertAlign);
 chroma_height = height >> 1;
 luma_size = width * height;
 chroma_size = width * chroma_height;
 mbi_size = (luma_size+chroma_size)/4;
 mbi_size = ((mbi_size + uMBIAlign) & ~ uMBIAlign);
```

3.3.2.2 VID_API_EVENT_SEQ_HDR_FOUND

Decoder finds one valid sequence header, and application can fetch sequence info from shared memory (MediaIPFW_Video_SeqInfo *)(pSharedInterface->SeqInfoTabDesc.pSeqInfoArrayBase) with stream index offset. The information includes the number of FS buffers required and the number of MBI buffers and size. The MBI buffers is slightly different depending on the codec format. For the DCP buffer, decoder should request a size of the worst-case for expected streams to support.

3.3.2.3 VID_API_EVENT_PIC_DECODED

One frame is decoded, and the application can fetch decoded picture information from the shared memory `(MediaIPFW_Video_PicInfo *)( pSharedInterface->PicInfoTabDesc.pPicInfoArray Base)` with stream index offset. Event data also contains the frame ID (uMsgData[0x7]) and frame location (uMsgData[0xA]) in the stream buffer.

Frame ID ‘0x555’ means skip frame, which is added mainly for synchronization and includes timestamp.

3.3.2.4 VID_API_EVENT_FRAME_BUFF_RDY

One frame is ready for display, and the application can fetch the picture information from the message data.

```
  uMsgData[0x0] = ulFsId;
  uMsgData[0x1] = ulFsLumaBase[0];
  uMsgData[0x2] = ulFsChromaBase[0] - ulFsLumaBase[0];
  uMsgData[0x3] = ulStride;
  uMsgData[0x4] = ulData;
  uMsgData[0x5] = ulFsLumaBase[0];
  uMsgData[0x6] = ulFsChromaBase[0];
  uMsgData[0x7] = ulFsLumaBase[1];
  uMsgData[0x8] = ulFsChromaBase[1];
```

Frame ID ‘0x555’ means to skip frame, which is added mainly for synchronization, including timestamp.

3.3.2.5 VID_API_EVENT_REL_FRAME_BUFF

Decoder sends this event to notify that the application's one frame won’t be referenced by decoder again. Application can fetch the frame information by converting ‘msgdata’ to MEDIA_PLAYER_FSREL.

```
typedef struct{
  u_int32   uFSIdx;
  MEDIAIP_MEM_REQ  eType;
  bool      bNotDisplayed;
} MEDIA_PLAYER_FSREL;
```

If the frame is also returned from render, the application can release the frame through command ‘VID_API_CMD_FS_RELEASE’ with uLocalCmdBuffer[0]=uFsID.

3.3.2.6 VID_API_EVENT_ABORT_DONE

Decoder responds for command VID_API_CMD_ABORT in seek/trick mode. To abort current playback to implement seek, application should insert some padding data (4K aligned) beginning with abort start code (4bytes aligned) manually without updating writer pointer.

Definition of abort start code (big endian) for different formats:

- AVC:0x0000010B
- VC1: 0x0000010A
- MPEG2: 0x000001B7
- MPEG4: 0x000001B1
- RV/VP6/VP8/SPARK: 0x00000134
- HEVC: 0x0000014A 0x20000000

3.3.2.7 VID_API_EVENT_STR_BUF_RST

Decoder responds for command VID_API_CMD_RST_BUF in seek/trick mode.

3.3.2.8 VID_API_EVENT_FINISHED

To finish the current playback, application should insert some padding data (4 KB aligned) beginning with EOS (end of stream) start code (4 bytes aligned) manually. After decoder finishes all process of these special EOS data, it reports the finish event to notify the application that all frames have been processed.

Definition of EOS start code (big endian) for different formats:

- AVC:0x0000010B
- VC1: 0x0000010A
- MPEG2: 0x000001CC
- MPEG4: 0x000001B1
- RV/VP6/VP8/SPARK: 0x00000134
- MJPEG:0x0000FFEF
- HEVC: 0x0000014A 0x20000000

3.3.2.9 VID_API_EVENT_STOPPED

Before the application releases resource and exits, the application should send the command ‘VID_API_CMD_STOP’ first, and then wait the event VID_API_EVENT_STOPPED from decoder. Suppose VID_API_CMD_STOP/VID_API_EVENT_STOPPED is only handled once for every playback. One reasonable flow for normal or exception exit (such as Ctrl-C) is application call VID_API_CMD_ABORT following VID_API_CMD_STOP.

3.3.2.10 VID_API_EVENT_FIRMWARE_XCPT

The firmware sends an exception event when decoder enters unrecoverable error state due to some unknown reasons. Additional information can be fetched from ‘(char*)msgdata’. In such a use case, the application should send ‘VID_API_CMD_FIRM_RESET’ to the software to reset the Cortex-M core and firmware.

3.3.3 Decoder State Machine

The following figure shows the state machine for decoder internal state, as well as the relation between commands and events.

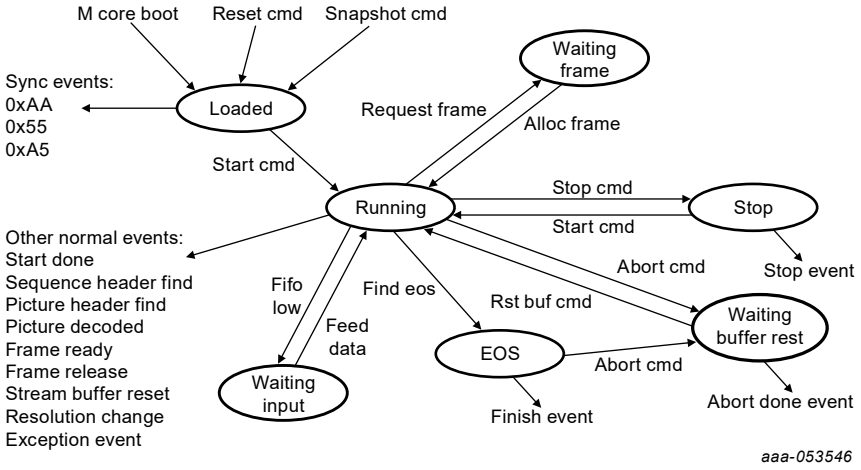

Figure 5. Decoder State Machine

3.3.4 Decoder Special Operation

There are some special operations beside normal playback for decoder.

3.3.4.1 Seek Mode

Playback will jump into one specified location from which one reference frame begins.

To support this mode, one suggested workflow is as follows:

1. Application fills data (4 KB aligned) beginning with ABORT start code (4 bytes aligned) following current write pointer, and does not update write pointer after padding data filling. It is very important that firmware updates write pointer of padding data instead of application, which will avoid potential risk caused by asynchronous operations.
2. Application sends the command ‘VID_API_CMD_ABORT’ with padding data size, e.g., cmdnum =1, and cmddata[0]=pading_size.
3. Application clears bit-stream buffer after receiving the event ‘VID_API_EVENT_ABORT_DONE’.
4. Application sends the command ‘VID_API_CMD_RST_BUF’.
5. Application feeds data from new location after receiving the event ‘VID_API_EVENT_STR_BUF_RST. During seek process, other events will be handled normally, such as VID_API_EVENT_REQ_FRAME_BUFF and VID_API_EVENT_REL_FRAME_BUFF.

3.3.4.2 Trick Mode

To speed up playback, application only feeds some reference frame and all non-reference frames will be dropped.

In such mode, decoder is requested to send out decoded frame immediately without any frame delay. One suggested workflow in such mode is as follows:

1. Application sends the command ‘VID_API_CMD_ABORT’.
2. Application clears bit-stream buffer after receiving the event ‘VID_API_EVENT_ABORT_DONE’.
3. Application sends the command ‘VID_API_CMD_RST_BUF’.
4. Application feeds one reference frame after receiving the event ‘VID_API_EVENT_STR_BUF_RST'.
5. Application fills data (4 KB aligned) beginning with EOS start code (4 bytes aligned) following the reference frame.
6. Application renders frame after receiving the event ‘VID_API_EVENT_FRAME_BUFF_RDY’.

7. Application waits until receiving the event ‘VID_API_EVENT_FINISHED’.
8. Loop from Step 1 to Step 7.

3.3.4.3 Low Latency Mode

In some special cases, only I and P frames are encoded and the display order is same as the decode order. To minimize the delay for frame buffer display, ‘low-latency’ mode is defined to support it.

Application is required to do the following two steps to enable it.

1. Disable reorder - Application should be responsible to guarantee that no reorder exists in the current clip. Set parameter ‘uDispImm’ to 1 in shared memory (MediaIPFW_Video_CodecParams*)pSharedInterface->CodecParamTabDesc.pCodecParamArrayBase with stream index offset.
2. Insert data (4 KB aligned) beginning with flush start code (4 bytes aligned) at the end of every frame. Definition of flush start code (only AVC supported): AVC: 0x00000115

3.3.4.4 Suspend and Resume Mode

Snapshot command is used to implement suspend/resume feature, which is OS and system related.

1. Applications send snapshot command first.
2. Application powers off the Cortex-M core after receiving snapshot done event (0xA5).
3. Application resumes decoder following re-power on the Cortex-M core in such cases. Configuring event ‘0xAA’ will be bypassed by firmware, and the application only receives started event ‘0x55’.

3.3.4.4.1 Limitation for Suspend and Resume

To avoid potential failure in suspend operation, the application should guarantee enough input data, and RPC variable ‘BUFFER_INFO_TYPE StreamBuffInfo[str_idx]’ is used to support this:

```
typedef struct {
  u_int32 stream_input_mode;
  u_int32 stream_pic_input_count;
  u_int32 stream_pic_parsed_count;
  u_int32 stream_buffer_threshold;
  u_int32 stream_pic_end_flag;
} BUFFER_INFO_TYPE, *pBUFFER_INFO_TYPE;
```

- stream_input_mode: 0: invalid; 1: frame-level; 2: non-frame level.
- stream_pic_input_count: accumulated frame count (set by application), meaningful only for frame-level mode. The application should be responsible to initialize (before start) and reset (abort done) it.
- stream_pic_parsed_count: accumulated frame count (set by firmware), meaningful only for frame-level mode.
- stream_buffer_threshold: the data threshold value to trigger internal HW preparser, meaningful only for non-frame level.
- stream_pic_end_flag: meaningful only for frame-level mode to support some special mode (such as trickmode). The application should set this ‘pic-end’ flag following all padding data inserted, including flush, EOS,abort. The application should clear this ‘pic-end’ flag after receiving the following events, including buffer reset done (abort) and finish (EOS) event. Clearing ‘pic-end’ for flush padding is not required, e.g., keep pic-end unchanged in low-latency mode.

3.4 Encoder Workflow

3.4.1 Stream Configuration

Applications need to set initial parameter for every stream (instance) before start. Sample code to initialize the encoder:

```
pVirt = (phy_to_virt)(pSharedInterface->pEncCtrlInterface[str_idx]);
pMEDIAIP_ENC_PARAM param = (phy_to_virt)(pVirt->pEncParam);
param->eCodecMode = MEDIAIP_ENC_FMT_H264;
param->uSrcStride = VPU_ENC_WIDTH_DEFAULT;
param->uSrcWidth = VPU_ENC_WIDTH_DEFAULT;
param->uSrcHeight = VPU_ENC_HEIGHT_DEFAULT;
param->uSrcOffset_x = 0;
param->uSrcOffset_y = 0;
param->uSrcCropWidth = VPU_ENC_WIDTH_DEFAULT;
param->uSrcCropHeight = VPU_ENC_HEIGHT_DEFAULT;
param->uOutWidth = VPU_ENC_WIDTH_DEFAULT;
param->uOutHeight = VPU_ENC_HEIGHT_DEFAULT;
param->uFrameRate = VPU_ENC_FRAMERATE_DEFAULT;
param->uMinBitRate = BITRATE_LOW_THRESHOLD
```

3.4.2 Encoder Event Handler

For encoding, the application should implement one event handler to respond the event from the Cortex-M core and send proper command to trigger correct encoder state. The following is one simple flow for normal encoder.

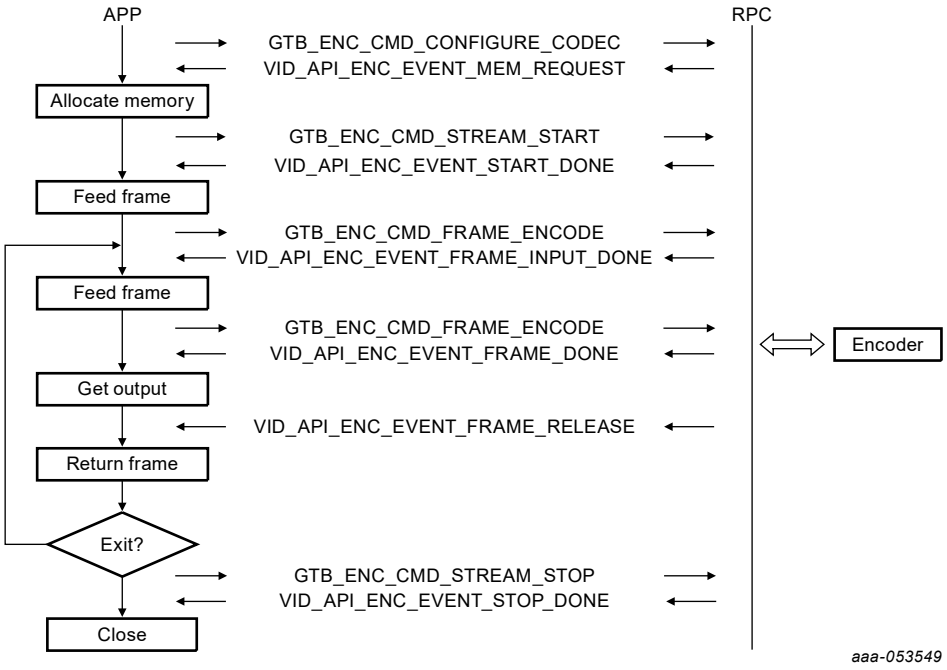

Figure 6. Simple flow for normal encoder

The input frame buffer is set through the buffer descriptor: `pSharedInterface -> pEncCtrlInterface [str_idx] ->pEncYUVBufferDesc`.

- Output frame buffer: Output data is maintained in buffer descriptor: `pSharedInterface-> pEncCtrlInterface[str_idx]-> pEncStreamBufferDesc`.
- Encoder parameters: The application can configure most parameters through set structure: `pSharedInterface-> pEncCtrlInterface[str_idx]->pEncParam` before sending the configuration command.

3.4.2.1 VID_API_ENC_EVENT_MEM_REQUEST

Encoder reports required memory information, and the application is responsible to allocate memory first, and then call the command 'GTB_ENC_CMD_STREAM_START' to start. The required memory size can be obtained from the message data.

Sample code to calculate total memory size:

```
MEDIAIP_ENC_MEM_REQ_DATA *req_data = msgdata;
for (i = 0; i < req_data->uEncFrmNum; i++) {
  size += req_data->uEncFrmSize;
}
for (i = 0; i < req_data->uRefFrmNum; i++) {
  size += req_data->uRefFrmSize;
}
size += req_data->uActBufSize;
```

The application is responsible to allocate the frame based on the information above and fill the structure (p MEDIAIP_ENC_MEM_POOL)pSharedInterface->pEncCtrlInterface[str_idx]->pEncMemPool accordingly.

3.4.2.2 VID_API_ENC_EVENT_START_DONE

Encoder is ready, and the application can send next input by filling 'pEncYUVBufferDesc' and call command 'GTB_ENC_CMD_FRAME_ENCODE'.

3.4.2.3 VID_API_ENC_EVENT_FRAME_INPUT_DONE

Encoder has obtained the frame input buffer, and the application can send next input by filling 'pEncYUVBufferDesc' and call command 'GTB_ENC_CMD_FRAME_ENCODE'.

3.4.2.4 VID_API_ENC_EVENT_FRAME_DONE

Encoder notifies the application that one frame is encoded, and the application can get output through 'pEncStreamBufferDesc'. Another way, the application can obtain picture information through '(pMEDIAIP_ENC_PIC_INFO)msgdata'. Start address of the output frame data is stored in the variable 'uStrBuffWrPtr', and the size of the output frame data is stored in the variable 'uFrameSize'.

3.4.2.5 VID_API_ENC_EVENT_FRAME_RELEASE

Encoder releases one input frame, and the application can get the frame ID through msgdata[0]. The application can refresh the input frame buffer data only after receiving this release event.

3.4.2.6 VID_API_ENC_EVENT_STOP_DONE

Response to the stop command, and the application can exit current stream after receiving this event.

3.4.2.7 VID_API_ENC_EVENT_FIRMWARE_XCPT

Firmware sends the exception event when encoder enters unrecoverable error state due to some unknown reasons. Additional information can be obtained from '(char*)msgdata'. In such cases, the application should send 'GTB_ENC_CMD_FIRM_RESET' to the software to reset the Cortex-M core and firmware.

3.4.3 Encoder State Machine

The following figure shows the state machine for encoder internal state, as well as the relation between commands and events.

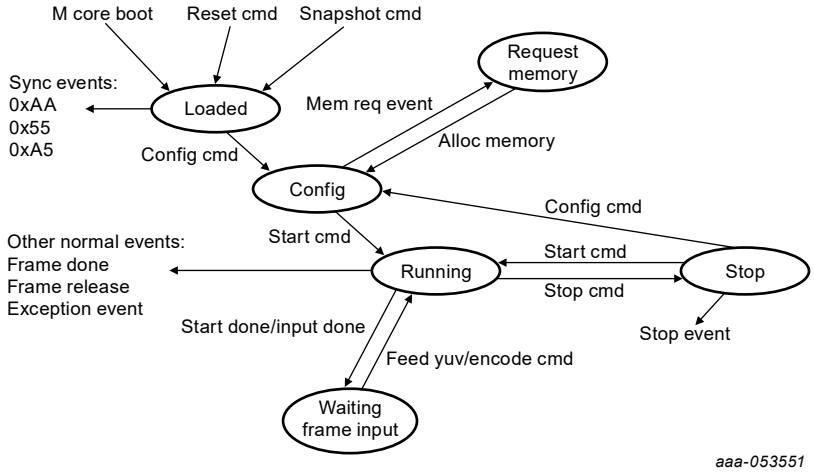

Figure 7. State machine for encoder internal state

3.4.4 Encoder Special Operations

There are some special operations beside normal encoding.

3.4.4.1 Low Latency Mode

Some delay is involved since encoder will consider frame re-order for more efficient compression, and one RPC parameter is added to allow application disable re-order feature to implement minimum latency:

```
((pMEDIAIP_ENC_PARAM)(pSharedInterface->pEncCtrlInterface[str_idx]->pEncParam))
 ->uLowLatencyMode
1: enable low-latency mode.
0: disable low-latency mode
```

3.4.4.2 Suspend and Resume

Snapshot command is used to implement suspend/resume feature, which is OS and system related.

1. Application sends the snapshot command.
2. Applications powers off the Cortex-M core after receiving the snapshot done event (0xA5).
3. Application resumes encoder following re-power on the Cortex-M core. In such cases, configuring event '0xAA' will be bypassed by firmware, and the application only receives the started event '0x55'.

3.5 Multi-instance Support

Decoder and encoder can support up to 8 multi-instances (streams). VID_API_NUM_STREAMS must be defined with 8. The application should obtain the real multi-instance number from the RPC variable reported from `'pSharedInterface->uMaxDecoderStreams'` after the decoder is loaded or from `'pSharedInterface->uMaxEncoderStreams'` after the encoder is loaded. There is stream index parameter in every RPC command and message to support multi-instance, and the application should be responsible to maintain context for every instance. The application can reuse the stream ID after the current stream is stopped.

3.6 Resolution Change

Different resolutions are permitted in some special clips or cases. For resolution change, decoder receives resolution change event 'VID_API_EVENT_RES_CHANGE' following new header event 'VID_API_EVENT_SEQ_HDR_FOUND'. The application should follow the rules below to support resolution change:

- Free and re-allocate all frame and MBI buffers for new resolution.
- Obtain the current sequence tag from 'uActiveSeqTag' in structure MediaIPFW_Video_SeqInfo for every header event 'VID_API_EVENT_SEQ_HDR_FOUND'.
- Pack sequence tag in the top 8 bits along with frame ID in the command 'VID_API_CMD_FS_RELEASE' and 'VID_API_CMD_FS_ALLOC':

```
uLocalCmdBuffer[0] = (ulFsId &0xFFF) | (uActiveSeqTag << 24)
```

3.7 Memory Requirements

3.7.1 Decoder Buffer

There are three types of buffers required based on the information reported from the structure MediaIPFW_Video_SeqInfo through the callback event 'VID_API_EVENT_SEQ_HDR_FOUND':

- **Frame buffers**: The number of frame buffer is decided by the variables 'uNumDPBFrms' and 'uNumRefFrms'. Each frame needs to be aligned due to the hardware limitation. Frame buffer size is equal to: Align(uHorRes,256) * Align(uVerRes ,256). Total number = uNumDPBFrms + uNumRefFrms.
- **MBI buffers**: The number of MBI buffer is decided by the frame buffer number, and the maximum number should be less than 19. One reasonable MBI buffer size is a quarter of one frame, and alignment is 2 KB (0x800).
- **DCP buffers**: The number of DCP buffer is reported by the variable 'uNumDFEAreas', and needs to reserve one fixed size (0x3000000: 48M) for each DCP buffer. The DCP buffer is used for parallel process and required only for the HEVC format.

3.7.2 Encoder Buffer

There are three types of buffers required based on the information reported from struct 'MEDIAIP_ENC_MEM_REQ_DATA' through the callback event 'VID_API_ENC_EVENT_MEM_REQUEST':

- Frame buffers: uEncFrmNum * frame buffers of size uEncFrmSize. uEncFrmNum is less than MEDIAIP_MAX_NUM_WINDSOR_SRC_FRAMES (defined with 6).
- Reference buffers: uRefFrmNum * frame buffers of size uRefFrmSize. uRefFrmNum is less than MEDIAIP_MAX_NUM_WINDSOR_REF_FRAMES (defined with 3).
- Activity buffer: An Activity buffer area of size uActBufSize. Activity buffer is accessed by M0+ core, so the application should guarantee that it is allocated in proper memory space. And 'uMemVirtAddr' is also required for the access by M0+.

3.7.3 Bitstream Buffer

The allocated bitstream buffer should hold at least 2 to 3 frames. To support decode for 4 KB HEVC clip, increase the bitstream buffer to 10 MB. To support encoder of 1080p H.264 clip, set the bitstream buffer to 2 MB or above.

3.7.4 YUV Input Frame Buffer

The application needs to maintain the frame input buffer queue, and 3 or above is suggested to improve the pipeline efficiency. Every frame buffer size is equal to width * height * 3/2.

3.7.5 RPC Decoder Shared Memory Size

The RPC memory size can be limited under 1 MB for 8 instances with the following details on the required size information:

- RPC Interface: sizeof(DEC_RPC_HOST_IFACE), max_str_num (max stream number) related
- Command ring buffer size (CMD_SIZE): recommended to reserve 20 KB
- Message ring buffer size (MSG_SIZE): recommended to reserve 20 KB
- Parameter buffer size (CODEC_SIZE): max_str_num * sizeof(MediaIPFW_Video_CodecParams)
- Jpeg parameter buffer size (JPEG_SIZE): max_str_num * sizeof(MediaIPFW_Video_JpegParams)
- Sequence buffer size (SEQ_SIZE): max_str_num * sizeof(MediaIPFW_Video_SeqInfo)
- Picture information size (PIC_SIZE): max_str_num * sizeof(MediaIPFW_Video_PicInfo)
- Gop information size (GOP_SIZE): max_str_num * sizeof(MediaIPFW_Video_GopInfo)
- Meter information size (QMETER_SIZE): max_str_num* sizeof(MediaIPFW_Video_QMeterInfo)
- Debug log buffer size (DBGLOG_SIZE): for internal debug data dump. Application can disable it through set ‘DbgLogDesc.uDecStatusLogSize = 0’
- Debug ring buffer size (DEBUG_SIZE): for debug log. Application can disable it through set uStart = uEnd
- Engineer access ring buffer size (ENG_SIZE): for internal engineer debug. Application can disable it through set uStart = uEnd
- Encrypt buffer size (reserved): max_str_num * sizeof(MediaIPFW_Video_Encrypt_Info)

3.7.6 RPC Encoder Shared Memory Size

The RPC memory size can be limited under 1 MB for 8 instances with the following details on the required size information:

- RPC Interface: sizeof(ENC_RPC_HOST_IFACE), max_str_num(max stream number) related
- Command ring buffer size (CMD_SIZE): recommended to reserve 20 KB
- Message ring buffer size (MSG_SIZE): recommended to reserve 20 KB
- Control interface: sizeof(MEDIA_ENC_API_CONTROL_INTERFACE), max_str_num related
- Other shared memory, max_str_num related:
  - Sizeof(MEDIAIP_ENC_YUV_BUFFER_DESC) //yuv input
  - Sizeof(BUFFER_DESCRIPTOR_TYPE) //bitstream buffer
  - Sizeof(MEDIAIP_ENC_EXPERT_MODE_PARAM) //reserved=
  - Sizeof(MEDIAIP_ENC_PARAM) //input parameters
  - Sizeof(MEDIAIP_ENC_MEM_POOL) //internal frame buffers
  - Sizeof(ENC_ENCODING_STATUS) //reserved
  - Sizeof(ENC_DSA_STATUS_t) //reserved

3.7.7 Firmware Size

For decoder in the current firmware release, the memory space is mapped into the following, so the application can reserve 32 MB for the whole binary space.

```
[.text]: 0 - 1MB
[.data/.bss/.stack]: 1MB - 31MB
```

For encoder the memory space is mapped into:

```
[.text]: 0 - 1MB
[.data/.bss/.stack]: 1MB - 2MB
e.g. application can reserve 2MB for the whole binary space.
```

3.7.8 Cortex-M Cores Memory Space

On the i.MX 8QuadXPlus/8QuadMax/8DualMax B0 SoC, the Cortex-M Cores are reserved for VPU, and the DDR memory space shared with the Arm cores is from 0x0 to 0x3FFFFFFF (1GB) for each Cortex-M core.

3.7.8.1 Memory Map Between Arm Core and Cortex-M Core

The 0x0 address is always reserved for the code segment, so the application needs to handle the shared memory (code binary and data buffer) carefully. For example, if the application allocates one memory to load firmware binary at 0x80000000, the Cortex-M core address offset register is set to 0x80000000, and then all data memory shared with the Cortex-M cores needs to be limited within the range (0x80000000, 0xC0000000).

3.7.8.2 Configuring Cached and Uncached Regions

In the Cortex-M cores, the memory cache map is defined as:

- 0x0000_0000 - 0x07FF_FFFF (128MB) cached (reserved for .text/.data)
- 0x0800_0000 - 0x0FFF_FFFF (128MB) uncached (reserved for shared memory)
- 0x1000_0000 - 0x1FFF_FFFF (256MB) cached (reserved for .text/.data)
- 0x2000_0000 - 0x3FFF_FFFF (512MB) uncached (reserved for shared memory)

To assign one section in the uncached region, keep its offset (compared with binary base address) in the range [128 MB, 256 MB] or [512 MB,1 GB].

3.7.8.3 Buffer Configuration Example

The following is an example for buffer configuration (binary address is set with offset 0):

```
Space [0, 128M]: Binary (include code/data/.bss): >32MB
Uncachable on Acore
Cachable on M core
Space [128M,256M]: RPC shared memory: > 2MB
Uncachable on Acore
Uncachable on M core
Space [256M,1G]: some other reserved buffers for M core if have
        Uncachable on Acore
        Cachable/Uncachable on M core  (cache property is unexpected)
```

For bit-stream and frame buffers, there is no space limitation, because they are referenced only by VPU hardware.

3.7.9 Platform and Cortex-M Core ID Configuration

To minimize the effort of the following maintain, one decoder binary is used to support all platforms for the i.MX 8QuadXPlus, 8QuadMax and 8DualMax. Some fixed memory offsets from binary base address are reserved to platform configuration.

```
Off[16]: Platform (0-QXP, 1-QM, 2-DM, 3-DX)
Off[17]: Mcore ID
QXP/DM: (0 or 1 for decoder ; 0 or 1 for encoder)
  QM:  (0, 1 or 2 for decoder ; 1 or 2 for encoder)
```

Example of Cortex-M core ID (default configuration on the Linux release):

- QXP/DM: 0 for decoder; 1 for encoder
- QM: 0 for decoder; 1 and 2 for two encoders separately

If the user changes the Cortex-M core ID, some register addresses should be changed accordingly, including CSR, MU base registers.

3.7.10 Boot Speedup

The Cortex-M core is responsible to clear all data in the `.bss` segment, but the Cortex-M core is much slower compared with the Arm core, so one additional memory offset is reserved to allow the firmware (Cortex-M core) to skip clearing of the .bss segment. In such cases, the application (Arm core) is required to clear related buffers (containing .bss) allocated for firmware.

```
  Off[20]: 1 (skip .bss clearing); 0 (default flow)
```

4 i.MX 6 VPU Main Features

The i.MX 6 VPU is fully compliant with H.264 BP/MP/HP, VC-1 SP/MP/AP, MPEG-4 SP/ASP except GMC, DivX (Xvid) and MPEG-1/2, VP8, AVS, and MJPEG. Image sizes up to full HD 1920x1080 60i or 30p decoding and 1920x1088 encoding. The VPU supports various error resilience tools, multiple decoding, and full duplex multi-party-call simultaneously. The VPU provides programmability, flexibility, and ease of upgrade in decoding and encoding or host interface because all of the controls in the decoding and encoding process and host interface are implemented as firmware in the programmable BIT processor.

The detailed features of the VPU are as follows:

- Encoding
  - H.264
    - 1/4-pel accuracy motion estimation with programmable search range up to [+/-128, +/-64]
    - Search range is reconfigurable by SW
    - 16x16, 16x8, 8x16 and 8x8 block sizes
    - Configurable block sizes
    - Only one reference frame for motion estimation
    - Intra-prediction
    - Luma I4x4 Mode : 9 modes
    - Luma I16x16 Mode : 3 modes (Vertical, Horizon, DC)

  - Chroma Mode : 3 modes (Vertical, Horizon, DC)
  - Minimum encoding image size is 96 pixels in horizontal and 16 pixels in vertical
  - FMO/ASO tool of H.264 is not supported
- MPEG-4
  - AC/DC prediction
  - 1/2-pel accuracy motion estimation with search range up to [+/-128, +/-64]
  - Search range is reconfigurable by SW
- H.263
  - H.263 Baseline profile + Annex J, K (RS=0 and ASO=0), and T
- 48x32 pixel minimum encoding image size (48 pixels horizontal and 32 pixels vertical)

- Decoding
- H.264
  - Fully compatible with the ITU-T Recommendation H.264 specification in BP/MP and HP
  - CABAC/CAVLC
  - Supports MVC Stereo High profile
  - Variable block size-16x16, 16x8, 8x16, 8x8, 8x4, 4x8 and 4x4
  - Error detection, concealment and error resilience tools
- VC1
  - All VC-1 profile features-SMPTE Proposed SMPTE Standard for Television: VC-1 Compressed Video Bitstream format and Decoding Process
  - Simple/Main/Advanced Profile
- MPEG-4
  - Simple/Advanced Simple profile except GMC
  - H.263 Baseline profile + Annex I, J, K (except RS/ASO), and T
  - DivX version 3.x to 6.x
  - Xvid
- MPEG-2
  - Fully compatible with ISO/IEC 13182-2 MPEG2 specification in main profile
  - I, P, and B frame
  - Field coded picture (interlaced) and frame coded picture
- AVS
  - Supports Jizhun profile level 6.2 (exclude 422 use case)
- VP8
  - Fully compatible with VP8 decoder specification
  - Supporting both simple and normal in-loop deblocking
- 64x64 pixel minimum decoding size

- JPEG tools
- MJPEG Baseline Process Encoder and Decoder
  - Baseline ISO/IEC 10918-1 JPEG compliance
  - Support 1 or 3 color components
  - 3 component in a scan (interleaved only)
  - 8 bit samples for each component
  - Support 4:2:0, 4:2:2, 2:2:4, 4:4:4 and 4:0:0 color format (max. six 8x8 blocks in one MCU)
  - Minimum encoding size is 16x16 pixels.

- Value added features
- De-ringing
- Pre/Post rotator/mirror

  - Built-in de-blocking filter for MPEG-2/MPEG-4 and DivX
- Programmability
  - 16-bit DSP processor dedicated to processing bitstream and controlling the codec hardware
  - General purpose registers and interrupt for communication to and from a host processor
- Optimal external memory accesses
  - Configurable frame buffer formats (linear or tiled) for longer burst-length
  - 2D cache for motion estimation and compensation to reduce external memory accesses
  - Secondary AXI port for on-chip memory to enhance performance
- Performance
  - All video decoder standards up to 1920x1088 @ 30 fps at 266 MHz
  - H264 encoder standards up to 1920x1088 @ 30 fps at 266 MHz, MPEG4 encoder up to 720p@30fps at 266MHz
  - MJPG decoder on 4:4:4 supports 120M pixel per second @ 266MHz
  - MJPG encoder on 4:4:4 supports 160M pixel per second @ 266MHz
- Interrupt
  - Interrupt from and to external host processor or interrupt controller

4.1 i.MX 6 VPU Programmability

The VPU has an internal DSP called the BIT processor which controls the internal hardware blocks for video decoder operations. The operation of the BIT processor is determined by the dedicated microcode called the BIT firmware. VPU has a complete set of BIT firmware code as well as a complete set of VPU control functions called VPU API. Therefore, application developers do not need to manage codec-specific issues on host processor.

4.1.1 Frame-Based Processing

The BIT processor completes decoding operations on a frame-by-frame basis, which allows low level independence of VPU operations from the host processor. While frame operations are running, there is no need for communication between the host processor and the VPU. Therefore, VPU does not burden the host processor during decoder operations.

After issuing a picture processing command, the host application performs its own operations until it is ready for the next picture processing operation or until it receives an interrupt from VPU informing the host processor of completion of the picture processing.

4.1.2 Program Memory Management

The VPU has its own program memory to load BIT firmware for supporting application-specific operations. In order to use this internal memory efficiently, the BIT firmware has a dynamic re-loading scheme which enables the VPU to have a small amount of program memory.

For example, if a MPEG-2 decoder operation is running on VPU, then VPU program memory is filled by the MPEG-2 decoder firmware inside VPU. If a H.264 decoder operation is newly issued, then the BIT processor automatically loads the H.264 decoder firmware from the SDRAM to program memory.

Because of the frame-based operation of VPU, the maximum rate of this dynamic reloading operation is approximately 30 times per second in a single instance decoder use case. Since the amount of BIT firmware for one decoder standard is smaller than 16 KB, this is not a large burden for the VPU operations in performance and memory bandwidth.

4.1.3 Multi-Instances

The VPU supports multiple instances which can be helpful for multi-channel decoder applications. In order to support this multi-instance operation, the BIT processor uses an internal context parameter set for each decoder instance. When creating a new instance and starting a picture processing operation, a set of context parameters is created and updated automatically within VPU. This internal context management scheme allows different decoder tasks running on the host processor to control VPU operations independently with their own instance numbers.

When creating a new instance, an application task receives a new handle specifying an instance if a new handle is available on the VPU. All the subsequent operations for the given application task are handled separately by VPU using this task-specific handle. When writing a VPU driver, this handle can be regard as a device-ID or a port-ID of the VPU for each task. Since the VPU can only perform one picture processing task at a time, the application task should check if VPU is ready before starting a new picture operation. An application can easily terminate a single task on VPU by calling a function for closing a certain instance.

4.2 i.MX 6 VPU Host Interface

This section describes the interfaces used by host processor to control i.MX 6 VPU.

This section presents a general description of the host interfaces provided for a host processor to control i.MX 6 VPU.

4.2.1 Communication Models

VPU requires a dedicated path for exchanging data and/or messages between the host processor and VPU. VPU uses shared memory for exchanging data between the host processor and VPU. This shared memory is accessible through ABMA host bus. Bitstream data and frame data are exchanged using this shared memory space.

Independent of data exchange path, a dedicated path for messages between the host processor and VPU is provided using a set of VPU registers called the host interface registers. All commands and responses between the host processor and VPU are exchanged through these registers as shown in the figure below.

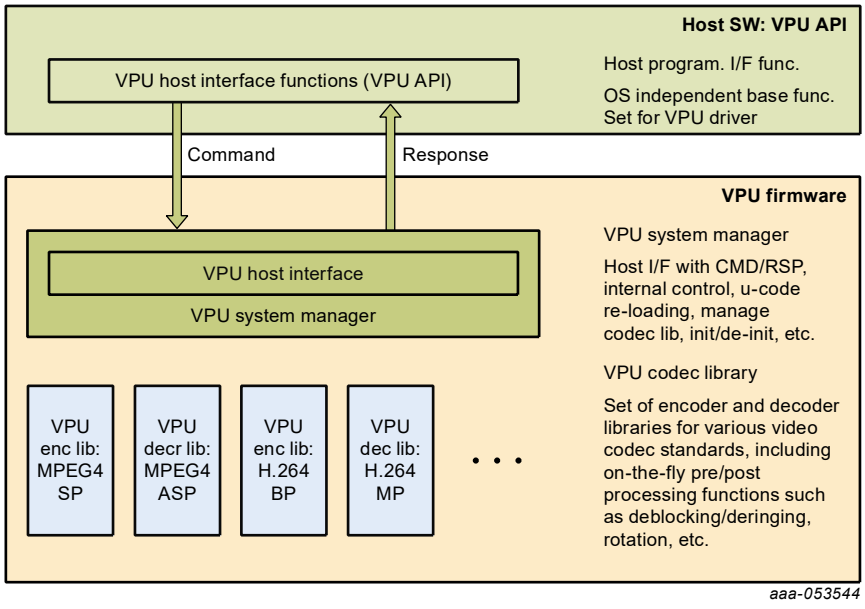

Figure 8. Data and Message Exchange Between Host and VPU

All bitstream and picture data is accessed directly by the host processor and VPU. The related information about the data transfer as well as command and responses is exchanged through the host interface. The host interface of the VPU uses a set of registers accessible from the host processor. Some of these host registers are used for exchanging actual command and responses and other registers are used to give information about the internal status of the VPU to host processor. Firmware running on the BIT processor is well-optimized for a given set of commands and responses.

4.2.1.1 Data Handling

All of the pixel data or stream data transactions are performed by the host processor or VPU through the shared memory space in SDRAM. In order to assure safe transactions between the host processor and VPU, all the required information is stored in the host interface registers. Generally, these transactions are one-directional transactions: the host or VPU writes the data and the other reads the data on a single data buffer. Therefore, transactions are easily and safely controlled by using a pair of read and write pointers.

Just as common data buffers in shared memory, the BIT processor requires a certain amount of memory for processing called the working buffer. The working buffer can only be accessed by VPU. In addition, frame buffers used in picture decoding are managed exclusively by VPU which ensures safe decoding.

For proper streaming, the available free space in the decoder stream buffer can be accessed using the buffer read pointer, write pointer, and buffer size. A set of APIs is provided for this purpose that can be called by the application anytime.

4.2.1.2 Host Interface Registers

A set of commands is provided for controlling codec operations on a frame-by-frame basis together with the corresponding responses. Host interface registers can be partitioned into three categories as follows:

- BIT processor control registers update or show BIT processor status to host processors. Most of these registers are used for initializing BIT processor during boot-up.
- BIT processor global registers store all the global variables which are reserved even while an active instance is changed. All the buffer addresses and some global options are safely stored in these registers.
- BIT processor command I/O registers are overwritten or updated whenever a new command is transmitted from the host processor. All commands with input arguments and all corresponding responses with return values are handled using these registers.

In addition, command I/O registers are used in a pre-defined way for each command to control VPU.

4.2.2 API-Based VPU Control

Host applications generally control VPU through a set of pre-defined APIs by sending a command and corresponding arguments to VPU. After receiving an interrupt from VPU, signaling the completion of the requested operation, the host application acquires the results as shown in the figure below.

Each API definition includes the requested command and the input and output data structure. The given command from the API function is always written on a dedicated I/O register, but the input and output data structure is transmitted through a set of command I/O registers that contain the input arguments and output results. Therefore, application developers do not need to know the details of the host register definitions and usage.

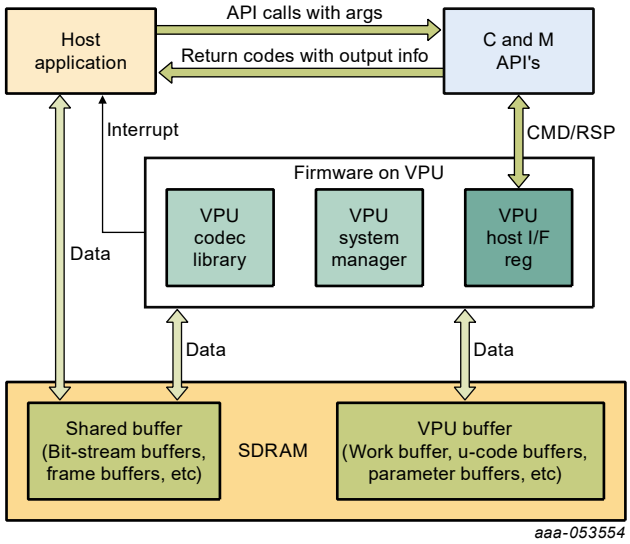

Figure 9. Software Control Model of VPU from Host Application

4.3 i.MX 6 VPU API Features

This section describes the important features of i.MX 6 VPU API, which is an API that includes a set of API functions to control the VPU.

A set of API functions is provided to efficiently control the VPU. The VPU API covers all functions of the i.MX 6 VPU. This API-based approach speeds up the development process of application software. Important features of the API for i.MX 6 VPU are summarized in the following sections.

4.3.1 Simple Software Control

The i.MX 6 VPU API provides a simple way to control the i.MX 6 VPU and avoid errors in application software. The host application does not need to know the details of the i.MX 6 VPU internal operations. For example, in order to initialize the VPU, an application simply calls API for initialization, vpu_Init(), and no additional information is required for calling this API. vpu_Init() API performs all the required steps for initializing the i.MX 6 VPU. When issuing a picture decoder operation, the application simply changes some variables included in the well-defined input data structure.

4.3.1.1 Handling Multi-Instances

The i.MX 6 VPU supports multiple instances for decoding and encoding at the same time, which can be used in multiple decoding and encoding and multi-party call applications. To support multi-instance operations, i.MX 6 VPU API provides a full set of functions for handling the instances with ease. When opening a new instance, the application receives a handle specifying the new instance provided a new handle is available at that time. The operations for a given instance are separately controlled using the corresponding handle. An application can easily terminate a single task on the VPU by calling a function for closing a certain instance.

4.3.1.2 Frame-Based Codec Processing

The i.MX 6 VPU completes decoding and encoding operation on a frame-by-frame basis, which enables low-level independence of the VPU operations from the host processor. While frame processing operation are running, there is no need for communication between the host processor and VPU. Therefore, the VPU does not burden the host processor during decoding and encoding operations.

4.4 Type Definitions

This section describes the types and structures used in the VPU API.

4.4.1 Type Definitions (common data types)

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

4.4.1.1 PhysicalAddress

```
typedef Uint32 PhysicalAddress;
```

Description

Represents physical addresses that are recognizable by VPU. In general, VPU hardware does not know about the virtual address space that is set and handled by the host processor. The virtual addresses are translated into physical addresses by the Memory Management Unit (MMU). Data buffer addresses, such as input bitstream buffer or frame buffer, are given to VPU as an address in the physical address space.

4.4.1.2 VirtualAddress

```
typedef Uint32 VirtualAddress;
```

Description

Represents virtual addresses that are recognizable by CPU.

4.4.1.3 CodStd

```
typedef enum {
                    STD_MPEG4 = 0,
                    STD_H263 = 1,
                    STD_AVC = 2,
                    STD_VC1 = 3,
                    STD_MPEG2 = 4,
                    STD_DIV3 = 5,
                    STD_RV = 6,
                    STD_MJPG = 7,
                    STD_AVS = 8,
                    STD_VP8 = 9,
} CodStd;
```

Description

Enumeration for declaring code standard type variables. The following video standards are supported by VPU:

- MPEG4 SP/ASP
- H.263 Profile 3
- AVC (H.264) BP/MP/HP
- VC-1 SP/MP/AP
- MPEG-2, MPEG-1
- Divx3
- AVS
- On2 VP8

Note: *The MPEG-1 decoder operation is handled as a special case of the MPEG-2 decoder.*

4.4.1.4 RetCode

```
typedef enum {
                    RETCODE_SUCCESS = 0,
                    RETCODE_FAILURE = -1,
                    RETCODE_INVALID_HANDLE = -2,
                    RETCODE_INVALID_PARAM = -3,
                    RETCODE_INVALID_COMMAND = -4,
                    RETCODE_ROTATOR_OUTPUT_NOT_SET = -5,
                    RETCODE_ROTATOR_STRIDE_NOT_SET = -11,
                    RETCODE_FRAME_NOT_COMPLETE = -6,
                    RETCODE_INVALID_FRAME_BUFFER = -7,
                    RETCODE_INSUFFICIENT_FRAME_BUFFERS = -8,
                    RETCODE_INVALID_STRIDE = -9,
                    RETCODE_WRONG_CALL_SEQUENCE = -10,
                    RETCODE_CALLED_BEFORE = -12,
                    RETCODE_NOT_INITIALIZED = -13,
                    RETCODE_DEBLOCKING_OUTPUT_NOT_SET = -14,
                    RETCODE_NOT_SUPPORTED = -15,
                    RETCODE_REPORT_BUF_NOT_SET = -16,
                    RETCODE_FAILURE_TIMEOUT = -17,
                    RETCODE_MEMORY_ACCESS_VIOLATION = -18,
                    RETCODE_JPEG_EOS = -19,
                    RETCODE_JPEG_BIT_EMPTY = -20
} RetCode;
```

Description

Enumeration for declaring the 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. Table 1 shows the basic meaning of each return code.

Table 1. Return Codes

Code	Description
RETCODE_SUCCESS	Operation successful
RETCODE_FAILURE	Operation not successfully; this value is returned when an un-recoverable decoder error occurs such as a header parsing error
RETCODE_INVALID_HANDLE	Given handle for current API function call is invalid, for example, not initialized yet or improper function call for the given handle
RETCODE_INVALID_PARAM	Given argument parameters (for example, input data structure) is invalid (not initialized yet or not valid anymore)
RETCODE_INVALID_COMMAND	Given command is invalid, for example, undefined or not allowed in the given instance
RETCODE_ROTATOR_OUTPUT_NOT_SET	Rotator output buffer is not allocated even though rotation is enabled
RETCODE_ROTATOR_STRIDE_NOT_SET	Rotator stride is not provided even though rotation is enabled
RETCODE_FRAME_NOT_COMPLETE	Frame decoding operation is not completed, so the given API function call is not allowed
RETCODE_INVALID_FRAME_BUFFER	Certain frame buffer pointers are invalid (not initialized yet or not valid)

Table 1. Return Codes*...continued*

Code	Description
RETCODE_INSUFFICIENT_FRAME_BUFFERS	Given numbers of frame buffers are not enough for the operations of the given handle. This return code is only received when calling the **DecRegisterFrameBuffer**() function
RETCODE_INVALID_STRIDE	Given stride is invalid (for example, 0, not a multiple of 8 or smaller than the picture size). This return code is only allowed in API functions which set stride
RETCODE_WRONG_CALL_SEQUENCE	Current API function call is invalid considering the allowed sequences between API functions (for example, missing one crucial function call before this function call)
RETCODE_CALLED_BEFORE	Multiple calls of current API function for a given instance are invalid
RETCODE_NOT_INITIALIZED	VPU is not initialized yet. Before calling any API functions, the initialization API function, **vpu_Init**(), should be called
RETCODE_DEBLOCKING_OUTPUT_NOT_SET	Not used in i.MX 6
RETCODE_NOT_SUPPORTED	One feature is not supported
RETCODE_REPORT_BUF_NOT_SET	Data report buffer address is not set with a valid value if report of MB, MV, frame status, slice information or user data is enabled
RETCODE_FAILURE_TIMEOUT	The hardware may be busy with other operation and unavailable for current API calling or something is wrong with VPU based. For detailed meaning of this return value, see each API description
RETCODE_MEMORY_ACCESS_VIOLATION	Memory access violation error
RETCODE_JPEG_EOS	The MJPEG bitstream comes to the end in the vpu_ DecStartOneFrame() API calling.
RETCODE_JPEG_BIT_EMPTY	The filled data in the bitstream buffer is not enough for the header parser in the vpu_DecStartOneFrame() API calling.

4.4.1.5 CodecCommand

```
typedef enum {
                    ENABLE_ROTATION,
                    DISABLE_ROTATION,
                    ENABLE_MIRRORING,
                    DISABLE_MIRRORING,
                    ENABLE_DERING,
                    DISABLE_DERING,
                    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_VIDEO_HEADER,
                    ENC_GET_VOS_HEADER,
                    ENC_GET_VO_HEADER,
                    ENC_GET_VOL_HEADER,
```

```
                    ENC_GET_JPEG_HEADER,
                    ENC_SET_INTRA_MB_REFRESH_NUMBER,
                    DEC_SET_DEBLOCK_OUTPUT,
                    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,
                    ENC_SET_REPORT_MBINFO,
                    ENC_SET_REPORT_MVINFO,
                    ENC_SET_REPORT_SLICEINFO,
                    DEC_SET_REPORT_BUFSTAT,
                    DEC_SET_REPORT_MBINFO,
                    DEC_SET_REPORT_MVINFO,
                    DEC_SET_REPORT_USERDATA,
                    SET_DBK_OFFSET,
                    SET_WRITE_MEM_PROTECT,
                    ENC_SET_SUB_FRAME_SYNC,
                    ENC_ENABLE_SUB_FRAME_SYNC,
                    ENC_DISABLE_SUB_FRAME_SYNC,
                    DEC_SET_FRAME_DELAY,
                    ENC_SET_INTRA_REFRESH_MODE,
                    ENC_ENABLE_SOF_STUFF
} CodecCommand;
```

Description

Special enumeration type for configuration commands from the host processor to VPU. Most of these commands are called occasionally (not periodically) for changing VPU operation configuration. Details of these commands are presented in Section 4.7.9 and Section 4.8.12.

Following commands aren't used on i.MX 6 platform:

```
SET_WRITE_MEM_PROTECT
ENC_SET_SUB_FRAME_SYNC
ENC_ENABLE_SUB_FRAME_SYNC
ENC_DISABLE_SUB_FRAME_SYNC
```

4.4.1.6 GDI_TILED_MAP_TYPE

```
typedef enum {
                    LINEAR_FRAME_MAP = 0,
                    TILED_FRAME_MB_RASTER_MAP = 1,
                    TILED_FIELD_MB_RASTER_MAP = 2,
                    TILED_MAP_TYPE_MAX
} GDI_TILED_MAP_TYPE;
```

Description

Enumeration type for the GDI type.

4.4.1.7 MirrorDirection

```
typedef enum {
                    MIRDIR_NONE,
                    MIRDIR_VER,
```

```
                    MIRDIR_HOR,
                    MIRDIR_HOR_VER
} MirrorDirection;
```

Description

Enumeration type for representing the mirroring direction.

4.4.1.8 Mp4HeaderType

```
typedef enum {
                    VOL_HEADER,
                    VOS_HEADER,
                    VIS_HEADER
} Mp4HeaderType;
```

Description

Special enumeration type for MPEG-4 top-level header classes such as visual sequence header, visual object header, and video object layer header.

4.4.1.9 AvcHeaderType

```
typedef enum {
                    SPS_RBSP,
                    PPS_RBS,
                    SPS_RBSP_MVC,
                    PPS_RBSP_MVC
} AvcHeaderType;
```

Description

Special enumeration type for AVC parameter sets such as sequence parameter set and picture parameter set.

4.4.1.10 EncHandle

```
typedef EncInst * EncHandle;
```

Description

Dedicated type for encoder handles returned when an encoder instance is opened. An encoder instance can be referred to by the corresponding handle. EncInst is a type managed internally by API and the application does not need to use it.

4.4.1.11 DecHandle

```
typedef DecInst * DecHandle;
```

Description

Dedicated type for decoder handles returned when a decoder instance is opened. A decoder instance can be referred to by the corresponding handle. DecInst is a type managed internally by API and the application does not need to use it.

4.4.2 Data and Structure Definitions

This section describes the data and structure definitions used in VPU API functions.

4.4.2.1 FrameBuffer

```
typedef struct {
                    Uint32 strideY;
                    Uint32 strideC;
                    int myIndex;
                    PhysicalAddress bufY;
                    PhysicalAddress bufCb;
                    PhysicalAddress bufCr;
                    PhysicalAddress bufMvCol;
} FrameBuffer;
```

Description

Data structure for representing frame buffer pointers for each color component

`strideY` is a Y stride value of the given frame buffers.

`strideC` is a C stride value of the given frame buffers.

`myIndex` is an A frame buffer index to identify each frame buffer that will be processed by VPU. The index of each buffer should be unique and less than 32.

`bufY` is an address for Y component in the physical address space.

`bufCb` is an address for Cb component in the physical address space.

`bufCr` is an address for Cr component in the physical address space.

`bufMvCol` is an address for co-located motion vector buffers in the physical address space.

The host application must allocate contiguous physical memory from SDRAM space for the components using this data structure. All four addresses must be 8-byte aligned. One pixel value of a component occupies one byte and the frame data is in YCbCr 4:2:0 format for H.264, H.264 and MPEG-4 codecs. The sizes of the Cb and Cr buffers are 1/4 the size of the Y buffer size for H.264, H.263 and MPEG-4 codecs. For MJPEG, the frame data format can be YCbCr 4:2:0, 4:2:2 horizontal, 4:2:2 vertical, 4:4:4 and 4:0:0 and the sizes of the Cb and Cr buffers vary. The co-located motion vector is only required for B-frame decoding in MPEG-2, AVC MP/HP, MPEG-4 ASP, VC-1 MP/AP, and so on.

4.4.2.2 DecMaxFrmInfo

```
typedef struct {
                    int maxMbX;
                    int maxMbY;
                    int maxMbNum;
} DecMaxFrmInfo;
```

Description

Data structure for representing maximum frame buffer info for decoder.

`maxMbX` means maximum supported macro blocks of horizontal direction.

`maxMbY` means maximum supported macro blocks of vertical direction.

`maxMbNum` means maximum supported macro blocks of one picture.

This structure is provided to the host application to specify maximum frame buffer information. Normally, without resolution change picture decoder support, `maxMbX` value is picture width/16, `maxMbY` is picture height/16, `maxMbNum` is width * height / 256. If the user knows there is a resolution change from smaller to bigger, the user must give the information as needed and allocate corresponding maximum frame buffer.

4.4.2.3 Rect

```
typedef struct {
                    Uint32    left;
                    Uint32    top;
                    Uint32    right;
                    Uint32    bottom;
} Rect;
```

Description

Data structure for representing a rectangular window in a frame.

`left` is a horizontal pixel offset of the top-left corner of the rectangle from top-left corner of the frame.

`top` is a vertical pixel offset of the top-left corner of the rectangle from top-left corner of the frame.

`right` is a horizontal pixel offset of the bottom-right corner of the rectangle from top-left corner of the frame.

`bottom` is a vertical pixel offset of the bottom-right corner of the rectangle from top-left corner of the frame.

This structure is provided to the host application to specify display window for the H.264 cropping option. Each value is offset from the start point of a frame. Therefore, all values are positive.

4.4.2.4 EncHeaderParam

```
typedef struct {
                    PhysicalAddress buf;
                    Uint8 *pBuf;
                    int size;
                    int headerType;
                    int userProfileLevelEnable;
                    int userProfileLevelIndication;
} EncHeaderParam;
```

Description

This structure is used for adding a header syntax layer to the encoded bit stream. The parameter `headerType` is the input parameter for VPU. The other two parameters are returned from VPU after completing the requested operation. If the encoder ringbuffer reset option is enabled, the parameters `buf` and `size` are also input parameters. In this situation, the host application must allocate the physical buffer to save the encoded header syntax to VPU.

`headerType` is the encode header code. In MPEG-4.

3'b000 - VOL header; 3'b001 - VOS header; 3'b010 - VO header

In H.264

3'b000 - SPS rbsp; 3'b001 - PPS rbsp

In H.263, `ENC_HEADER` command is ignored.

`userProfileLevelEnable` decides whether to set profile_and_level_indication in VOS header as MPEG-4 predefined values. If `UserProfileLevelEnable` is 0, profile_and_level_indication is encoded with one of these values:

8'b0000 0001 : L1 <= 176x144@15Hz

8'b0000 0010 : L2 <= 352x288@15Hz

8'b0000 0011 : L3 <= 352x288@30Hz

8'b0000 0100 : L4a <=640x480@30Hz

8'b0000 0101 : L5 <=720x576@25Hz

8'b0000 0110 : L6 <= otherwise

If `UserProfileLevelEnable` is 1, a host can set user profile and level with UserProfileLevelIndication.

`UserProfileLevelIndication` is a user-defined profile and level value for profile_and_level_indication in VOS.

4.4.2.5 EncParamSet

```
typedef struct {
                    Uint32 *paraSet;
                    Uint8 *pParaSet;
                    int size;
} EncParamSet;
```

Description

This is a structure used when the host processor requires SPS or PPS data from an encoder instance. The resulting SPS or PPS data is used in an application as a type of out-of-band information.

`paraSet` is the address of the SPS or PPS data.

`pParaSet` is the address of the MJPG encoder header data. It is only for MJPG.

`size` is the size of the data.

4.4.2.6 EncMp4Param

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

Description

This is the data structure for configuring MPEG4-specific parameters in encoder applications.

`mp4_dataPartitionEnable` where 0 = disable, 1 = enable

`mp4_reversibleVlcEnable` where 0 = disable, 1 = enable

`mp4_intraDcVlcThr` is the value of `intra_dc_vlc_thr` in MPEG-4 part 2 standard. Valid range is 0-7.

`mp4_hecEnable` where 0 = disable, 1 = enable.

`mp4_verid` is the value of MPEG-4 part 2 standard version ID. Both version 1 and 2 are allowed.

4.4.2.7 EncH263Param

```
typedef struct {
                    int h263_annexIEnable;
                    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_annexIEnable` where 0 = disable, 1 = enable . Not in use for i.MX 6 .

`h263_annexJEnable` where 0 = disable, 1 = enable

`h263_annexKEnable` where 0 = disable, 1 = enable

`h263_annexTEnable` where 0 = disable, 1 = enable

4.4.2.8 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;
                    int avc_fmoSliceSaveBufSize;
                    int avc_frameCroppingFlag;
                    int avc_frameCropLeft;
                    int avc_frameCropRight;
                    int avc_frameCropTop;
                    int avc_frameCropBottom;
                    int mvc_extension;
                    int interview_en;
                    int paraset_refresh_en;
                    int prefix_nal_en;
                    int avc_vui_present_flag;
                    VuiParam avc_vui_param;
                    int avc_level;
} EncAvcParam;
```

Description

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

`avc_constrainedIntraPredFlag` where 0 = disable, 1 = enable

`avc_disableDeblk` where 0 = enable, 1 = disable, 2 = disable deblocking filter at slice boundaries

`avc_deblkFilterOffsetAlpha deblk_filter_offset_alpha` (-6 to 6)

`avc_deblkFilterOffsetBeta deblk_filter_offset_beta` (-6 to 6)

`avc_chromaQpOffset chroma_qp_offset` (-12 to 12)

`avc_audEnable` where 0 = disable, 1 = enable. The encoder generates AUD RBSP at the start of every picture.

`avc_fmoEnable` is not used in the i.MX 6 since FMO encoding is not supported.

`avc_fmoSliceNum` is not used in the i.MX 6 since FMO encoding is not supported.

`avc_fmoType` is not used in the i.MX 6 since FMO encoding is not supported.

`avc_fmoSliceSaveBufSize` is not used in the i.MX 6 since FMO encoding is not supported.

`avc_frameCroppingFlag` where 0 = disable, 1 = enable. If this is 1, the encoder will generate frame_cropping_flag syntax at the SPS header.

`avc_frameCropLeft` is the sample number of left cropping region in a line.

`avc_frameCropRight` is the sample number of right cropping region in a line.

`avc_frameCropTop` is the sample number of top cropping region in a picture column.

`avc_frameCropBottom` is the sample number of bottom cropping region in a picture column.

`mvc_extension` where 0 = AVC, not MVC, 1 = MVC

`interview_en` where 0 = disable, 1 = enable interview prediction for another picture.

`paraset_refresh_en` 0 = disable, 1 = enable to insert SPS/PPS before anchor picture.

`prefix_nal_en` where 0 = disable, 1 = enable to add prefix nal unit before every 2nd view of MVC stream.

4.4.2.9 EncMjpgParam

```
typedef struct {
                    int mjpg_sourceFormat;
                    int mjpg_restartInterval;
                    int mjpg_thumbNailEnable;
                    int mjpg_thumbNailWidth;
                    int mjpg_thumbNailHeight;
                    Uint8 * mjpg_hufTable;
                    Unit8 * mjpg_qMatTable;
                    Uint8 huffVal[4][162];
                    Uint8 huffBits[4][256];
                    Uint8 qMatTab[4][64];
                    Uint8 cInfoTab[4][6];
} EncMjpgParam;
```

Description

This is a data structure for configuring MJPEG-specific parameters in encoder applications.

`mjpg_sourceFormat` is the chroma format. The format means chrominance size of source image and can be a value between 0 and 4: 0 = 4:2:0, 1 = 4:2:2 horizontal, 2 = 4:2:2 vertical, 3 = 4:4:4, 4 = 4:0:0.

`mjpg_restartInterval` is the value for representing interval of restart marker in MB unit.

`mjpg_thumbNailEnable` where 0 = disable, 1 = enable and the encoder enables thumbnail encoding.

`mjpg_thumbNailWidth` is the variable representing the width (in pixels) of the thumbnail to be encoded. This variable can have a value between 0 and the source image width. This value must be larger than a specific value and must be a multiple of the value shown in table below.

Table 2. mjpg_thumbNailWidth and mjpg_thumbNailHeight Values

Format	Value
4:2:0	16
4:2:2	16
2:2:4	8
4:4:4	8
4:0:0	8

`mjpg_thumbNailHeight` is the variable representing the width (in pixels) of the thumbnail to be encoded. This variable can have a value between 0 and the source image width. This value must be larger than a specific value and must be a multiple of the value shown in table above.

`mjpg_qMatTable` is the variable representing a pointer to an address in the Q-Matrix.

`mjpg_hufTable` is the variable representing a pointer to an address in the Huffman table (not used in i.MX 6).

huffVal[4][162] A list of the 8-bit symbol values in Huffman tables

huffBits[4][256] A 16-byte list giving the number of codes for each code length from 1 to 16 in Huffman tables.

qMatTab[4][64] Quantization tables

cInfoTab[4][6] Component information tables

4.4.2.10 EncSliceMode

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

Description

This is a structure used for declaring encoder slice mode and its options. This structure value is ignored for a MJPEG encoder.

`sliceMode` where 0 = one slice per picture, 1 = multiple slices per picture. In normal MPEG-4 mode, the `resync-marker` and packet header are inserted between slice boundaries. In short video header with Annex K = 0, the 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` is the size of a generated slice when sliceMode = 1, 0 means `sliceSize` is defined by amount of bits, and 1 means sliceSize is defined by MB(macro block) in a slice. This parameter is ignored when sliceMode = 0 or in short video header mode with Annex K = 0.

`sliceSize` is the size of a slice in bits or MB specified by sliceSizeMode. This parameter is ignored when sliceMode = 0, or in short video header mode with Annex K = 0.

4.4.2.11 EncOpenParam

```
typedef struct {
                    PhysicalAddress bitstreamBuffer;
                    Uint32 bitstreamBufferSize;
                    CodStd bitstreamFormat;
                    int picWidth;
```

```
                    int picHeight;
                    Uint32 frameRateInfo;
                    int bitRate;
                    int initialDelay;
                    int vbvBufferSize;
                    int gopSize;
                    int linear2TiledEnable;
                    int mapType;
                    EncSliceMode slicemode;
                    int intraRefresh;
                    int sliceReport;
                    int mbReport;
                    int mbQpReport;
                    int rcIntraQp;
                    int chromaInterleave;
                    int dynamicAllocEnable;
                    int ringBufferEnable;
                    union {
                              EncMp4Param mp4Param;
                              EncH263Param h263Param;
                              EncAvcParam avcParam;
                              EncMjpgParam mjpgParam;
                    } EncStdParam;
                    int userQpMin;
                    int userQpMax;
                    int userQpMinEnable;
                    int userQpMaxEnable;
                    Uint32 userGamma;
                    int RcIntervalMode;
                    int MbInterval;
                    int avcIntra16x16OnlyModeEnable;
                    int MESearchRange;
                    int MEUseZeroPmv;
                    int IntraCostWeight;
} EncOpenParam;
```

Description

This is a data structure for parameters when an encoder instance is opened.

`bitstreamBuffer` is a start address of bit stream buffer into which encoder places the bit streams. This address must be 512 byte-aligned.

`bitstreamBufferSize` is the size in bytes of a buffer pointed to by bitstreamBuffer. This value must be a multiple of 1024. The maximum size is 16383x1024 bytes.

`bitstreamFormat` is the standard type of bitstream in encoder operation: STD_MPEG4, STD_H263, STD_AVC, STD_VP8, STD_AVS or STD_MJPG.

`picWidth` is the width of a picture to be encoded in pixels.

`picHeight` is the height of a picture to be encoded in pixels.

`frameRateInfo` is the 16 least significant bits, [15:0] is a numerator and 16 most significant bits, [31:16] is a denominator for calculating the frame rate. The numerator is clock ticks per second and the denominator is clock ticks between frames minus 1. The frame rate can be defined by (numerator/(denominator + 1)), which equals (frameRateInfo & 0xffff) /((frameRateInfo >> 16) + 1). For example, a frameRateInfo value of 30 represents 30 frames/sec and the value 0x3e87530 represents 29.97 frames/sec.

`bitRate` is the target bit rate in kbps. If 0, there is no rate control and pictures are encoded with a quantization parameter equal to quantParam in EncParam. For MJPEG, this value is ignored. Users can control the MJPEG compression rate by setting qMatTab[4][64] of Section 4.4.2.9.

`initialDelay` is a time delay (in ms) 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.

`vbvBufferSize vbv_buffer_size` in bits. This value is ignored if rate control is disabled or initialDelay is 0. The value 0 means the encoder does not check for reference decoder buffer size constraints.

`gopSize` is the GOP size where 0 = only first picture is I, 1 = all I pictures, 2 = IPIP, 3 = IPPIPP, and so on. The maximum value is 32,767, but in practice, a smaller value should be chosen by the application for proper error concealment. This value is ignored for STD_MJPG.

`linear2TiledEnable` where 0 = disable, 1 = enable to convert linear to tiled format in vpu

`mapType` where 0 = Linear frame map; 1 = Frame tiled map; 2 = Field tiled map

`slicemode` where parameter for slice mode

`intraRefresh` where 0 = Intra MB refresh is not used. Otherwise = At least *N* MB's in every P-frame are encoded as intra MB's. This value is ignored in for STD_MJPG.

`sliceReport` is not used in i.MX 6 .

`mbReport` is not used in i.MX 6 .

`mbQpReport` is not used in i.MX 6 .

`rcIntraQp` is the quantization parameter for I frame. When this value is -1, the quantization parameter for I frames is automatically determined by VPU. In MPEG4/H.263 mode, the range is 1-31. In H.264 mode, the range is from 0-51. This is ignored for STD_MJPG.

`dynamicAllocEnable` is not used in i.MX 6 .

`ringBufferEnable` where 0 = disable, 1 = enable. This flag enables the streaming mode for the 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 using this flag. When this field is set, packet-based streaming with ring-buffer is used. When this field is not set, frame-based streaming with line-buffer is used.

`mp4Param` is a parameter for MPEG-4 part 2 Visual.

`h263Param` is a Parameter for ITU-T H.263.

`avcParam` is a parameter for AVC.

`mjpgParam` is a parameter for MJPEG.

`userQpMin` sets the minimum quantized step parameter for encoding process. -1 disables this setting and VPU uses the default minimum quantize step (Qp(H.264 12, MPEG-4/H.263 2). In MPEG-4/H.263 mode, the value of `userQpMix` is in the range of 1 to 31 and less than `userQpMax`. In H.264 mode, the value of `userQpMix` is in the range of 0 to 51 and less than `userQpMax`.

`userQpMax` sets the maximum quantized step parameter for the encoding process. -1 disables this setting and VPU uses the default maximum quantized step. In MPEG-4/H.263 mode, the value of `userQpMax` is in the range of 1 to 31. In H.264 mode, the value of `userQpMax` is in the range of 0 to 51. `userQpMin` and `userQpMax` must be set simultaneously.

`userQpMinEnable userQpMinEable` equal to 1 indicates that macroblock QP, generated in rate control, is cropped to be bigger than, or equal to, `userQpMin`.

`userQpMaxEnable userQpMaxEable` equal to 1 indicates that macroblock QP, generated in rate control, is cropped to be smaller than, or equal to, `userQpMax`.

`userGamma` is a smoothing factor in the estimation. A value for gamma is factorx32768, where the value for factor must be between 0 and 1. If the smoothing factor is close to 0, Qp changes slowly. If the smoothing factor is close to 1, Qp changes quickly. The default Gamma value is 0.75x32768.

`RcIntervalMode` is an encoder rate control mode setting. The host sets the bitrate control mode according to the required use case. The default value is 1. 0 = normal mode rate control 1 = FRAME_LEVEL rate control 2 = SLICE_LEVEL rate control 3 = USER DEFINED MB LEVEL rate control.

`MbInterval` is a user defined Mbyte interval value. The default value is 2 macroblock rows. For example, if the resolution is 720x470, then the two macroblock row is 2x(720/16) = 90. This value is used only when the RcIntervalMode is 3.

`avcIntra16x16OnlyModeEnable` is not used in i.MX 6.

`MESearchRange` is the search range mode for Motion Estimation.

0 : Horizontal(-128 ~ 127), Vertical(-64 ~ 63)

1 : Horizontal(-64 ~ 63), Vertical(-32 ~ 31)

2 : Horizontal(-32 ~ 31), Vertical(-16 ~ 15)

3 : Horizontal(-16 ~ 15), Vertical(-16 ~ 15)

`MEUseZeroPmv` is the PMV option for motion estimation. If this field is 1, encoding quality could be worse than when it was zero.

0 : Motion Estimation engine uses PMV that was derived from neighbor MV

1 : Motion Estimation engine uses Zero PMV

`IntraCostWeight` is the intra cost weight factor for Intra/Inter type decision. By default, it could be zero. If this register have some value W, and the cost of best intra mode that was decided by Refine-Intra-Mode-Decision is ICOST, the Final Intra Cost FIC will be like this, FIC = ICOST + W. So, if this field is not zero, the Final Intra Cost have additional weight. Then the Intra/Inter mode decision logic tend to make more Inter-Macroblock.

4.4.2.12 EncReportBufSize

```
typedef struct {
                    int sliceInfoBufSize;
                    int mbInfoBufSize;
                    int mvInfoBufSize;
} EncReportBufSize;
```

Description

This is a data structure to get the data report buffer size to start encoding from the encoder. Then the application allocates the memory according to the size information from the data report.

`sliceInfoBufSize` is a buffer size for slice information.

`mbInfoBufSize` is a buffer size for MB information.

`mvInfoBufSize` is a buffer size for motion vector information.

4.4.2.13 EncInitialInfo

```
typedef struct {
```

```
                    int minFrameBufferCount;
                    EncReportBufSize reportBufSize;
} EncInitialInfo;
```

Description

This is a data structure for parameters of `vpu_EncGetInitialInfo()` which are needed to get the initial information for encoder.

`minFrameBufferCount` is a minimum required buffer count in host applications. This returned value is used to allocate frame buffers in `vpu_EncRegisterFrameBuffer()`.

`reportBufSize` is the data report requested buffer size information.

4.4.2.14 EncParam

```
typedef struct {
                    FrameBuffer * sourceFrame;
                    int encTopOffset;
                    int encLeftOffset;
                    int forceIPicture;
                    int skipPicture;
                    int quantParam;
                    PhysicalAddress picStreamBufferAddr;
                    int picStreamBufferSize;
                    int enableAutoSkip;
} EncParam;
```

Description

This is a data structure for configuring one frame encoding:

`encTopOffset` is the top offset for cropping from source image to be encoded.

`encLeftOffset` is the left offset for cropping from source image to be encoded.

`sourceFrame` is a frame buffer containing source image to be encoded.

`forceIPicture`. If this value is 0, the picture type is determined by the VPU according to the various parameters such as encoded frame number and GOP size. If this value is 1, the frame is encoded as an I-picture regardless of the frame number or GOP size and I-picture period calculation is reset to the initial state. For MPEG-4 and H.263, I-picture is sufficient for decoder refresh. For H.264 mode, the picture is encoded as an Instantaneous Decoding Refresh (IDR) picture. This value is ignored if skipPicture = 1.

`skipPicture`. If this value is 0, the encoder encodes the picture normally. If this value is 1, the encoder ignores sourceFrame and generates a skipped picture. In this situation, the reconstructed image is a duplication of the previous picture. The skipped picture is encoded as P-type regardless of GOP size.

`quantParam` is used for all quantization parameters with VBR (no rate control). The range of value is 1-31 for MPEG-4 and 0-51 for H.264. When rate control is enabled, this field is ignored.

`picStreamBufferAddr` is a 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, an application might use multiple picture stream buffers for the best performance. Using this variable, an application re-registers the start position of the picture stream while issuing a picture encoding operation. This start address of this buffer must be 8-byte aligned. Its size is specified by `picStreamBufferSize`. In packet-based streaming with ring-buffer, this variable is ignored. This variable is only meaningful when both line-buffer mode and dynamic buffer allocation are enabled.

`picStreamBufferSize` is a byte size of a picture stream chunk. This variable represents byte size of a picture stream buffer and is crucial in line-buffer mode because encoder output can be corrupted if this size is smaller than any picture encoded output. Therefore, this value should be big enough for storing multiple picture streams with average size. In packet-based streaming with ring-buffer, this variable is ignored. This variable specifies the picture stream buffer size for encoded output in line-buffer mode.

`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).

4.4.2.15 EncReportInfo

```
typedef struct {
                    int enable;
                    int type;
                    int size;
                    Uint8 *addr;
} EncReportInfo;
```

Description

This is a structure used for reporting encoder information.

`enable` is a data report enabled or disabled; type, size and addr are valid when this flag is 1.

`type` is a type of mvInfo or sliceInfo.

`size` is a data report size.

`addr` is a saved report information address.

ReportInfo

```
typedef       struct {
                    int enable;
                    int size;
                    Uint32 *addr;
                    union {
                              int mvNumPerMb;
                              int userDataNum;
                              int type;
                    };
                    union {
                              int userDataBufFull;
                              int reserved;
                    };
} ReportInfo;
```

Description

This is a data structure for reporting the encoding/decoding information.

enable 0 - Disable the information report; 1- Enable the information report;

size The size of the buffer pointed by the addr to save the specific returned information while calling the vpu_DecGiveCommand or vpu_EncGiveCommand to set the buffer while it's used as input parameter. The size of returned information that's saved in the buffer pointed by the *addr* after vpu_DecGetOutputInfo or vpu_EncGetOutputInfo calling while it's used as output parameter. The *size* has different meanings for different information report cases. For MV information report in decoding, it represents the total number of MB.

addr The base address of the buffer to save the specific information.

mvNumPerMb When the MB report feature is enabled, decoder will report the motion vector into the buffer pointed by addr. mvNumPerMb is motion vector number of an macrobloc (MB). the size above is the total number of macroblock. Therefore total size of MB information in byte is size* mvNumPerMb *4.

userDataNum When the user data report is enabled, decoder will report user data content into the buffer pointed by addr. The size is the size of user data in byte. When user data report mode is 1 and user data size is bigger than user data buffer size, VPU reports user data as much as buffer size, skips the remains and sets userDataBufFull.

type When the MV report is enabled in the encoder, this value is used for picture type reporting in MVInfo.

userDataBufFull While user data report is enabled and the user data size is too small to save all decoded user data, VPU will set userDataBufFull as 1.

reserved Used by driver internally, host application should never use it.

4.4.2.16 EncOutputInfo

```
typedef struct {
                    PhysicalAddress bitstreamBuffer;
                    Uint32 bitstreamSize;
                    int bitstreamWrapAround;
                    int skipEncoded;
                    int picType;
                    int numOfSlices;
                    int reconFrameIndex;
                    Uint32 *pSliceInfo;
                    Uint32 *pMBInfo;
                    Uint32 *pMBQpInfo;
                    EncReportInfo mbInfo;
                    EncReportInfo mvInfo;
                    EncReportInfo sliceInfo;
} EncOutputInfo;
```

Description

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

`bitstreamBuffer` is a physical address of the starting point of a newly encoded picture stream. If dynamic buffer allocation is enabled in line-buffer mode, this value is identical to the picture stream buffer address specified by the host application.

`bitstreamSize` is a byte size of the encoded bitstream.

`bitstreamWrapAround` is a flag for bitstream buffer wrap-around. When this flag is set, the bitstream buffer wrapped around and a larger buffer size is required.

`skipEncoded`. 0 means current frame was encoded as non-skipped frame. 1 means current Frame was encoded as skipped frame.

`picType` is a picture type of the current decoded picture. This value has different meaning for different codecs: for VC1 SP/MP: 0 = I picture, 1 = P picture, 2 = BI picture, 3 = B picture, 4 = SKIPPED picture. For VC1 AP interlacing, picType contains two picture type information fields: bit[2:0] and bit[5:3] and the respective value has same meaning as SP/MP use case: 0 = I picture, 1 = P picture, 2 = BI picture, 3 = B picture, 4 = SKIPPED picture. For example, 0 = 000_000: both first and second field are I picture, 1 = 000_001: first field is I picture and second field is P picture In other codec use cases, 0 = I picture, 1 = P picture, 2 = B picture.

`numOfSlices` is a number of slices included in the newly encoded picture. When `sliceReport` in `EncOpenParam` is 0, this value is invalid.

`pSliceInfo` is not used in the i.MX 6 .

`pMBInfo` is not used in the i.MX 6 .

`pMBQpInfo` is not used in the i.MX 6 .

`mbInfo` is the MB information in the encoded picture. If the application does not give the `ENC_SET_REPORT_MBINFO` command to enable it before starting one frame encoding, this information is invalid.

`mvInfo` is a motion vector information in the encoded picture. If the application does not give the `ENC_SET_REPORT_MVINFO` command to enable it before starting one frame encoding, this information is invalid.

`sliceInfo` is a slice information in the encoded picture. If the application does not give the `ENC_SET_REPORT_SLICEINFO` command to enable it before starting one frame encoding, this information is invalid.

4.4.2.17 SearchRamParam

```
typedef struct {
                    PhysicalAddress searchRamAddr;
                    int SearchRamSize;
} SearchRamParam;
```

Description

This is not used in the i.MX 6 .

4.4.2.18 DecParamSet

```
typedef struct {
                    Uint32 * paraSet;
                    int size;
} DecParamSet;
```

Description

Structure used when the host processor requires to send SPS data or PPS data. The SPS data or PPS data is used in real applications as a type of out-of-band information.

4.4.2.19 DecOpenParam

```
typedef struct {
                    CodStd bitstreamFormat;
                    PhysicalAddress bitstreamBuffer;
                    Uint8 *pBitStream;
                    int bitstreamBufferSize;
                    int qpReport;
                    int mp4DeblkEnable;
                    int reorderEnable;
                    int chromaInterleave;
                    int filePlayEnable;
                    int picWidth;
                    int picHeight;
```

```
                    int avcExtension;
                    int dynamicAllocEnable;
                    int streamStartByteOffset;
                    int mjpg_thumbNailDecEnable;
                    PhysicalAddress psSaveBuffer;
                    int psSaveBufferSize;
                    int mp4Class;
                    int mapType;
                    int tiled2LinearEnable;
                    int bitstreamMode;
                    int jpgLineBufferMode;
} DecOpenParam;
```

Description

This is a data structure used to open a new decoder instance:

`bitstreamFormat` is a standard type of bitstream in decoder operation. One of codec standards defined in Section 4.4.1.3.

`bitstreamBuffer` is a start physical address of bit stream buffer from which the decoder retrieves the next bitstream. This address must be 512 byte-aligned.

`bitstreamBufferSize` is a size in bytes of a buffer pointed by bitstreamBuffer. This value must be a multiple of 1024. The maximum size is 16383x1024 bytes.

`qpReport` is not used in the i.MX 6 .

`mp4DeblkEnable` where 0 = disable, 1 = enable. In MPEG4 and H.263 (post-processing) modes, the decoder applies MPEG-4 deblocking filtered output to the host application.

`reorderEnable` where 1 = enables display buffer reordering when decoding H.264 streams. In H.264 mode, the output decoded picture is re-ordered if pic_order_cnt_type is 0 or 1 and the decoder must delay the output display for re-ordering. However, some applications (such as video telephony) do not require such display delay. The host may set this flag to 0 to disable output display buffer reordering. Then the BIT processor does not re-order the output buffer when pic_order_cnt_type is 0 or 1. If pic_order_cnt_type is 2 or in MPEG4 or H.263 modes, this flag is ignored because output display buffer reordering is not allowed.

`chromaInterleave` where 0 = CbCr not interleaved, 1 = CbCr interleaved.

`filePlayEnable` is not used in the i.MX 6 .

`picWidth` is a horizontal picture size read from the file format header used for codecs for which the picture size is not available in the bitstream, for example DivX3.11.

`picHeight` is a vertical picture size read from the file format header used for codecs for which the picture size is not available in the bitstream, for example DivX3.11.

`avcExtension` where 0 = no extension of AVC, 1 = MVC extension of AVC.

`dynamicBuffAllocEnable` is not used in the i.MX 6 .

`streamStartByteOffset` is a start byte offset of the stream buffer. Since the VPU has an internal limitation that the stream buffer start address must be 8-byte aligned, the host application may be required to copy the stream data to an 8-byte aligned buffer. This offset allows this overhead to be saved. The values should be between 0 and 7.

`mjpg_thumbNailDecEnable` is not used in the i.MX 6 .

`psSaveBuffer` is a start address of the PS (SPS/PPS) save buffer which the decoder saves PS (SPS/PPS) RBSP. This address must be 8 byte-aligned. This variable is only valid for H.264 decoder mode.

`psSaveBufferSize` is a size in bytes of a buffer pointed to by psSaveBuffer. This value must be a multiple of 1024. The maximum size is 65565x1024 bytes. This variable is only valid when decoding H.264 streams.

`mp4Class` is a MPEG4 class when codec is MPEG4 type 0 = MPEG-4; 1 = DivX 5.0 or higher; 2 = Xvid; 5 = DivX 4.0

`mapType` is an A Map type for GDI interface. 0 is a linear frame map. 1 is a frame tiled map. 2 is a filed tiled map.

`tiled2LinearEnable` is a tiled to linear map enable mode. The map type can be changed from tiled to linear in the post processing unit for display.

`bitstreamMode`. When read pointer reaches write pointer in the middle of decoding one picture. 0 means VPU sends an interrupt to HOST and waits for more bitstream to decode. (interrupt mode). 1 means VPU returns to the status right before the PIC_RUN command (rollback mode).

`jpgLineBufferMode` where 0 is a LineBuffer mode and 1 is a streaming mode.

4.4.2.20 DecReportBufSize

```
typedef struct {
                    int frameBufStatBufSize;
                    int mbInfoBufSize;
                    int mvInfoBufSize;
} DecReportBufSize;
```

Description

Not used in the i.MX 6 .

4.4.2.21 DecInitialInfo

```
typedef struct {
                    int picWidth;
                    int picHeight;
                    Uint32 frameRateInfo;
                    Uint32 frameRateRes;
                    Uint32 frameRateDiv;
                    Rect picCropRect;
                    int mp4_dataPartitionEnable;
                    int mp4_reversibleVlcEnable;
                    int mp4_shortVideoHeader;
                    int h263_annexJEnable;
                    int minFrameBufferCount;
                    int frameBufDelay;
                    int nextDecodedIdxNum;
                    int normalSliceSize;
                    int worstSliceSize;
                    int mjpg_thumbNailEnable;
                    int mjpg_sourceFormat;
                    int streamInfoObtained;
                    int profile;
                    int level;
                    int interlace;
                    int constraint_set_flag[4];
                    int direct8x8Flag;
                    int vc1_psf;
                    int aspectRateInfo;
                    Uint32 errorcode;;
```

```
                    int bitRate;
                    Vp8ScaleInfo vp8ScaleInfo;
                    int mjpg_ecsPtr;
                    DecReportBufSize reportBufSize;
                    AvcVuiInfo avcVuiInfo;
} DecInitialInfo;
```

Description

This is a data structure to get information necessary to start decoding:

`picWidth` is a horizontal picture size in pixels. This width value is used when allocating decoder frame buffers. In some situations, this returned value, the display picture width declared on the stream header, should be modified before allocating the frame buffers. When the picture width is not a multiple of 16, the picture width for buffer allocation should be re-calculated from the declared display width as: picBufWidth = ((picWidth + 15)/16) x 16, where picBufWidth is the horizontal picture buffer width. When picWidth is a multiple of 16, picWidth = picBufWidth.

`picHeight` is a vertical picture size in pixels. This height value is used when allocating decoder frame buffers. In some situations, this returned value, the display picture height declared on the stream header, should be modified before allocating the frame buffers. When the picture height is not a multiple of 16, the picture height for buffer allocation should be re-calculated from the declared display height as: picBufHeight = ((picHeight + 15)/16) x 16, where picBufHeight is the vertical picture buffer height. When picHeight is a multiple of 16, picHeight = picBufHeight.

`frameRateInfo` is not used in the i.MX 6 .

`frameRateRes` is the numerator part of frame rate fraction. Refer to DecOutputInfo.frameRateRes.

`frameRateDiv` is the denominator part of frame rate fraction. Refer to DecOutputInfo.frameRateDiv.

`picCropEnable` indicates if `picCropRect` is valid. If `picCropEnable` = 0,the picCropRect should be ignored. `picCropEnable` = 1, there is cropping rectangle information `picCropRect`.

`picCropRect` is a picture cropping rectangle information. If `picCropEnable` = 0, this field is invalid. This structure specifies the cropping rectangle information only for a H.264 decoder. The size and position of the cropping window in a full frame buffer is presented in this structure. This structure is only valid for H.264 decoder mode.

`mp4_dataPartitionEnable` where 0 = disable. 1 = enable.

`mp4_reversibleVlcEnable` where 0 = disable. 1 = enable.

`mp4_shortVideoHeader` where 0 = disable. 1 = enable.

`H263_annexJEnable` where 0 = disable. 1 = enable.

`minFrameBufferCount` is a minimum number of frame buffers required for decoding. The application must allocate at least this number of frame buffers and register those number of buffers to the VPU using `vpu_DecRegisterFrameBuffer()` before decoding pictures.

`frameBufDelay` is a maximum display frame buffer delay for buffering decoded picture reorder. The VPU may delay decoded picture displays for display reordering H.264 mode, when pic_order_cnt_type is 0 or 1 and for B-frame handling in VC-1 decoder. (By default, some H.264 encoder set pic_order_cnt_type to 0 or 1, but in BP applications, this setting is not actually used in practice.)

`nextDecodedIdxNum` is a maximum number of indexes which are returned after decoding one frame. the VPU may return 1 for MPEG-4, H.264, DivX, and MPEG-2 use cases. For VC-1 decoding only, this variable may have a value between 1 and 3.

`normalSliceSize` is a recommended size of buffer to save slice in normal use case. Value is determined by a quarter of the memory size of one raw YUV image in Kbytes.

`worstSliceSize` is a recommended size of buffer used to save slice in worst case. Value is determined by half of the memory size for one raw YUV image in Kbytes.

`mjpg_thumbNailEnable` where 0 = disable. 1 = enable. The stream which is decoded as thumbnail.

`mjpg_sourceFormat` is the chroma format of encoded image of the stream. The format defines the chrominance size of the source image and can be a value between 0 and 4. 0 = 4:2:0, 1 = 4:2:2 horizontal, 2 = 4:2:2 vertical, 3 = 4:4:4, 4 = 4:0:0

`streamInfoObtained`. Set to zero so the stream information cannot be obtained in the current firmware. It is true always on i.MX 6 .

`profile` is the profile information in the stream. This value is used as outlined below:

- H.264 : profile_idc
- Vc1 : 0~2 (SMTPE reserved), 3(advanced profile)
- MP2 : 3'b101: Simple, 3'b100: Main, 3'b011: SNR Scalable, 3'b10: Spatially Scalable, 3'b001: High
- MP4 : If VOS header is existed, 8'b00000000: Simple Profile, 8'b00001000: Advanced coding efficiency; 8'b00001111: Advanced Simple Profile
- If there is only VOL header, 8'b00000001: Simple Profile, 8'b00001100: Advance coding efficiency, 8'b00010001: Advanced Simple Profile

`level` is the level information in the stream. This value is used as outlined below:

- H.264 : level_idc
- Vc1 : level
- MP2 : 4'b1010: Low, 4'b1000: Main, 4'b0110: High 1440, 4'b0100: High
- MP4 : If VOS header is existed (high bit is 1, 8'b10000000), 4'b0000 or 4'b1000: L0, 4'b0001: L1, 4'b0010: L2, 4'b0011: L3...; If there is VOS header, level cannot be obtained.

`interlace` is the interlace information in the stream where 0 means only progressive frames in the stream, and 1 means there may be interlaced frame in the stream.

`constraint_set_flag` is a syntax element in H.264 used to make level in H.264. Ignored in other standards.

`direct8x8Flag` is a H.264 SPS syntax element which is used in B picture.

`vc1_psf` is a PSF information in VC1 stream information.

`aspectRateInfo` is an aspect rate information in stream information. If the value is 0, then aspect ratio information is not present.

- [H.264] - if aspectRateInfo [31:16] is 0, aspectRateInfo [7:0] means aspect_ratio_idc. Otherwise, AspectRatio means Extended_SAR.
- sar_width = aspectRateInfo [31:16]
- sar_height = aspectRateInfo [15:0]
- [VC-1]- Aspect Width = aspectRateInfo [31:16]
- Aspect Height = aspectRateInfo [15:0]
- [MP4] - This value is the index of Table 6-12 in ISO/IEC 14496-2.
- [MP2] - This value is the index of Table 6-3 in ISO/IEC 13818-2. It is determined by half of the memory size for one raw YUV image in KB unit.

`reportBufSize` is a data report requested buffer size information.

`bitRate` is the bitrate value written in bitstream syntax. Available only when value is not 1.

`vp8ScaleInfo` is VP8 up-sampling information. Refer to the Vp8ScaleInfo.

`mjpg_ecsPtr` is the consumed mjpg size for using software GetInitialInfo for MJPG decoder.

4.4.2.22 ExtBufCfg

```
typedef struct {
                    PhysicalAddress bufferBase;
                    int bufferSize;
} ExtBufCfg
```

Description

This data structure is used when the host application wants to give external memory configuration to VPU.

`bufferBase` is the start address of external memory.

`bufferSize` is the size of the buffer pointed by bufferBase in bytes.

4.4.2.23 DecBufInfo

```
typedef struct {
                    ExtBufCfg avcSliceBufInfo;
                    ExtBufCfg vp8MbDataBufInfo;
                    DecMaxFrmInfo   maxDecFrmInfo;
} DecBufInfo;
```

Description

This data structure is used when the host application wants to transfer additional buffer information without the frame buffer.

`avcSliceBufInfo` is the start address and size of the slice save buffer where decoder can save slice RBSP. This variable is only valid for H.264 decoder.

`vp8MbDataBufInfo` is the start address and the size of macroblock prediction data save buffer in which the VP8 decoder can save inflated macroblock information for a frame. This buffer is temporal scratch memory that sustains while decoding a picture. The start address must be 8-byte aligned.

`maxDecFrmInfo` is the maximum supported info of the frame buffer. Not used in the i.MX 6 .

4.4.2.24 DecParam

```
typedef struct {
                    int prescanEnable;
                    int prescanMode;
                    int dispReorderBuf;
                    int iframeSearchEnable;
                    int skipframeMode;
                    int skipframeNum;
                    int chunkSize;
                    int picStartByteOffset;
                    PhysicalAddress picStreamBufferAddr;
                    int mjpegScaleDownRatioWidth;  /* mx6 */
                    int mjpegScaleDownRatioHeight;  /* mx6 */
                    PhysicalAddress phyJpgChunkBase;
                    unsigned char *virtJpgChunkBase;
}DecParam;
```

Description

This is a data structure for picture decoding options:

`prescanEnable` is not used in the i.MX 6 .

`prescanMode` is not used in the i.MX 6 .

`iframeSearchEnable` where 0 = disable, 1 = enable, and the decoder performs skipping frame decoding until decoder meets an I (IDR) frame. If there is no I frame in the stream, the decoder waits for a I (IDR) frame. If skipframeNum is n, the decoder seeks the $(n + 1)^{th}$ I (IDR) frame. When decoder meets an EOS (End Of Sequence) code during I-Search, the decoder returns -1 (0xFFFF). If this option is enabled, `skipframeMode` options are ignored.

`skipframeMode` is a skip frame function enable and operation mode. 0 means skip frame disable, 1 means skip frame enabled (skip frames but I (IDR) frame), 2 means skip frame enabled (skip any frames). If this option is enabled, the decoder skips decoding as far as skipframeNum frames. After the decoder skips frames, the decoder returns decoded index -2 (0xFFFE) when it does not have any frames displayed. When decoder meets EOS (End Of Sequence) code during frame skip, the decoder returns -1 (= 0xFFFF).

`skipframeNum` is not used in the i.MX 6.

`chunkSize` is not used in the i.MX 6 .

`picStartByteOffset` is not used in the i.MX 6 .

`picStreamBufferAddr` is not used in the i.MX 6 .

`mjpegScaleDownRatioWidth` is horizontal down-sampling factor. 0 : No scaling, 1 : 1/2 down-scaling, 2 : 1/4 down-scaling, 3 : 1/8 down-scaling.

`mjpegScaleDownRatioHeight` is vertical down-sampling factor. 0 : No scaling, 1 : 1/2 down-scaling, 2 : 1/4 down-scaling, 3 : 1/8 down-scaling.

`phyJpgChunkBase` is the physical memory address of input bitstream buffer for Jpg.

`virtJpgChunkBase` is the point of virtual memory address of input bitstream buffer for Jpg.

4.4.2.25 DecReportInfo

```
typedef struct {
                    int enable;
                    int size;
                    union {
                              int mvNumPerMb;
                              int userDataNum;
                    };
                    union {
                              int reserved;
                              int userDataBufFull;
                    };
                    Uint8 *addr;
} DecReportInfo;
```

Description

This function is not used in the i.MX 6 .

4.4.2.26 Vp8ScaleInfo

```
typedef struct {
                    unsigned hScaleFactor : 2;
                    unsigned vScaleFactor : 2;
                    unsigned picWidth     : 14;
```

```
                    unsigned picHeight    : 14;
} Vp8ScaleInfo;
```

Description

This is data structure of picture up-scaling information for post-processing out of decoding loop.

This structure is valid only for VP8 decoding use case and can never be used by VPU itself. If host has an up sampling device, this information is useful. When the host allocates a frame buffer, application needs up-scaled resolution derived by this information to allocate enough (maximum) memory for variable resolution picture decoding.

`hScaleFactor` is an up-scaling factor for horizontal expansion. The value could be 0 to 3. The meaning of each value is described below:

- 0 means 1 up-sampling ratio
- 1 means 5/4 up-sampling ratio
- 2 means 5/3 up-sampling ratio
- 3 means 2/1 up-sampling ratio.

`vScaleFactor` is an up-scaling factor for vertical expansion. The value could be 0 to 3. The meaning of each value is described below:

- `picWidth` is a picture width in units of sample.
- `picHeight` is a picture height in units of sample.

4.4.2.27 Vp8PicInfo

```
typedef struct {
                    unsigned showFrame     : 1;
                    unsigned versionNumber : 3;
                    unsigned refIdxLast    : 8;
                    unsigned refIdxAltr    : 8;
                    unsigned refIdxGold    : 8;
} Vp8PicInfo;
```

Description

This is a data structure for VP8-specific header information and reference frame indices. Only VP8 decoder returns this structure after decoding a frame.

`showFrame` is the frame header syntax which means whether the current decoded frame is displayable or not. It is 0 when current frame is not for display and 1 when current frame is for display.

`versionNumber` is the VP8 profile version number information in the frame header. The version number enables or disables certain features in bitstream. It can be defined with one of the four different profiles: 0 to 3. Each indicates different decoding complexity.

`refIdxLast` is the frame buffer index for the Last reference frame. This field is valid only for next inter frame decoding.

`refIdxAltr` is the frame buffer index for the altref (Alternative Reference) reference frame. This field is valid only for next inter frame decoding.

`refIdxGold` is the frame buffer index for the Golden reference frame. This field is valid only for next inter frame decoding.

4.4.2.28 AvcFpaSei

```
typedef struct {
                    unsigned exist;
                    unsigned frame_packing_arrangement_id;
                    unsigned frame_packing_arrangement_cancel_flag;
                    unsigned quincunx_sampling_flag;
                    unsigned spatial_flipping_flag;
                    unsigned frame0_flipped_flag;
                    unsigned field_views_flag;
                    unsigned current_frame_is_frame0_flag;
                    unsigned frame0_self_contained_flag;
                    unsigned frame1_self_contained_flag;
                    unsigned frame_packing_arrangement_ext_flag;
                    unsigned frame_packing_arrangement_type;
                    unsigned content_interpretation_type;
                    unsigned frame0_grid_position_x;
                    unsigned frame0_grid_position_y;
                    unsigned frame1_grid_position_x;
                    unsigned frame1_grid_position_y;
                    unsigned frame_packing_arrangement_repetition_period;
} AvcFpaSei;
```

Description

This is a data structure for AVC FPA (frame packing arrangement) SEI.

0 means AVC FPA SEI does not exist. 1 means AVC FPA SEI exists.

`frame_packing_arrangement_id 0 ~ 2^32-1` is an identifying number that may be used to identify the usage of the frame packing arrangement SEI message.

`frame_packing_arrangement_cancel_flag` indicates whether the frame packing arrangement SEI message cancels the persistence of any previous frame packing arrangement SEI message in output order.

`quincunx_sampling_flag` indicates whether each color component plane of each constituent frame is quincunx sampled.

`spatial_flipping_flag` indicates that one of the two constituent frames is spatially flipped.

`frame0_flipped_flag` indicates which one of the two constituent frames is flipped.

`field_views_flag 1` indicates that all pictures in the current coded video sequence are coded as complementary field pairs.

`current_frame_is_frame0_flag` indicates the current decoded frame and the next decoded frame in output order.

`frame0_self_contained_flag` indicates whether inter prediction operations within the decoding process for the samples of constituent frame 0 of the coded video sequence refer to samples of any constituent frame 1.

`frame1_self_contained_flag` indicates whether inter prediction operations within the decoding process for the samples of constituent frame 1 of the coded video sequence refer to samples of any constituent frame 0.

`frame_packing_arrangement_extension_flag 0` indicates that no additional data follows within the frame packing arrangement SEI message.

`frame_packing_arrangement_type` is the type of packing arrangement of the frames as specified in Table D-8, ISO/IEC 14496-10D.2.25.

`content_interpretation_type` indicates the intended interpretation of the constituent frames.

`frame0_grid_position_x` specifies the horizontal location of the upper left sample of constituent frame 0 to the right of the spatial reference point.

`frame0_grid_position_y` specifies the vertical location of the upper left sample of constituent frame 0 below the spatial reference point.

`frame1_grid_position_x` specifies the horizontal location of the upper left sample of constituent frame 1 to the right of the spatial reference point.

`frame1_grid_position_y` specifies the vertical location of the upper left sample of constituent frame 1 below the spatial reference point.

`frame_packing_arrangement_repetition_period` indicates persistence of the frame packing arrangement SEI message.

4.4.2.29 MvcPicInfo

```
typedef struct {
                    int viewIdxDisplay;
                    int viewIdxDecoded;
} MvcPicInfo;
```

Description

This is a data structure for MVC-specific picture information. Only MVC decoder returns this structure after decoding a frame.

`viewIdxDisplay` is the view index order of display frame buffer corresponding to indexFrameDisplay of DecOutputInfo structure.

`viewIdxDecoded` is the view index order of decoded frame buffer corresponding to indexFrameDecoded of DecOutputInfo structure.

4.4.2.30 DecOutputInfo

```
typedef struct {
                    int indexFrameDisplay;
                    int indexFrameDecoded;
                    int NumDecFrameBuf;
                    int picType;
                    int picTypeFirst;
                    int idrFlg;
                    int numOfErrMBs;
                    Uint32 *qpInfo;
                    int hScaleFlag;
                    int vScaleFlag;
                    int indexFrameRangemap;
                    int prescanresult;
                    int notSufficientPsBuffer;
                    int notSufficientSliceBuffer;
                    int decodingSuccess;
                    int interlacedFrame;
                    int mp4PackedPBframe;
                    int h264Npf;
                    int pictureStructure;
                    int topFieldFirst;
                    int repeatFirstField;
                    union {
                              int progressiveFrame;
```

```
                            int vc1_repeatFrame;
                    };
                    int fieldSequence;
                    int decPicHeight;
                    int decPicWidth;
                    Rect decPicCrop;
                    int aspectRateInfo;
                    Uint32 frameRateRes;
                    Uint32 frameRateDiv;
                    Vp8ScaleInfo vp8ScaleInfo;
                    Vp8PicInfo vp8PicInfo;
                    MvcPicInfo mvcPicInfo;
                    AvcFpaSei avcFpaSei;
                    AvcVuiInfo avcVuiInfo;
                    int frameStartPos;
                    int frameEndPos;
                    int consumedByte;
                    DecReportInfo mbInfo;
                    DecReportInfo mvInfo;
                    DecReportInfo frameBufStat;
                    DecReportInfo userData;
} DecOutputInfo;
```

Description

This is a data structure to get information resulting from decoding a frame.

`indexFrameDisplay` is a frame buffer index of a picture to be displayed among frame buffers which were registered using `vpu_DecRegisterFrameBuffer()`. Frame data to be displayed is stored into the frame buffer specified by this index. When a delay in display does not exist, this index is the same as indexFrameDecoded. But if not, (for example, display reordering in AVC or B-frames in VC-1), this index is not the same value as indexFrameDecoded. If the decoder cannot provide a display output at the beginning of sequence decoding with different display order, this index always has -2 (0xFFFE) or -3 (0xFFFD) depending on the decoder skip option. And at the end of sequence decoding, if there is no more output for display, this value has -1 (0xFFFF). By checking this index, the host application can easily know whether sequence decoding has finished or not.

`indexFrameDecoded` is a frame buffer index of decoded picture among frame buffers which were registered using `vpu_DecRegisterFrameBuffer()`. A decoded frame during current picture decoding operation is stored into the frame buffer specified by this index. If decoder meets EOS or skip, the decoder returns -1 (0xFFFF) to represent that no decoded output is generated. Because of delays in display, the return value of -1 does not mean end of decoding. In order to check the end of decoding, the host application should refer to`indexFrameDisplay`.

`picType` is a picture type of the decoded picture where 0 = I picture, 1 = P picture, 2 = B picture. For H.264, bit[0] indicates IDR frame. 0 means current frame is IDR. 1 means non-IDR frame. If 0, the bit [2:1] should be ignored. If 1 of bit [0], bit [2:1] represents the slice types of current picture. 0 means I-slice, 1 means P-slice, 2 means B-slice. The actual value is the value of the ORed value of all slices of the current picture.

`numOfErrMBs` is a number of erroneous macroblocks while decoding a picture.

`qpInfo` is not used in the i.MX 6.

`hScaleFlag` is a flag for reduced resolution output in horizontal direction. For VC1 decoding, the resulting picture width from the decoder may be half the decoded picture width. In this situation, this flag is set. The host application should scale up the picture by two times in the horizontal direction to get proper display output.

`vScaleFlag` is a flag for reduced resolution output in vertical direction. For VC1 decoding, the resulting picture height from the decoder may be half the decoded picture height. In this situation, this flag is set. The host application should scale up this picture by two times in the vertical direction to get proper display output.

`indexFrameRangemap` is not used in the i.MX 6.

`prescanResult` is not used in the i.MX 6.

`notSufficientPsBuffer` is a flag that represents whether PS (SPS/PPS) save buffer is sufficient to decode the current picture. VPU does not get the last part of the current picture stream because of the buffer overflow. The host must close the current instance since the picture streams cannot be decoded properly because of loss of SPS/PPS data.

`notSufficientSliceBuffer` is a flag that represents whether slice save buffer is sufficient to decode the current picture. VPU does not get the last part of the current picture stream, and macroblock errors issue because of buffer overflow. The host can continue decoding the remaining pictures of the current input stream without closing the current instance, even though several pictures can be error-corrupted.

`decodingSuccess` where bit 0 = 0 means incomplete finish of decoding process and bit 0 = 1 means complete finish of decode process. This variable means that the decoding process is finished completely. If stream has errors in the picture header syntax or the first slice header syntax of H.264 stream, VPU does not initiate the MB decoding routine and returns immediately. In this situation, VPU returns bit 0 = 0, which means incomplete end of decoding process. Additionally, this variable uses some bits to indicate error reasons why VPU returns from picture decoding. In rollback mode, if bitstream buffer doesn't have enough bits for decoding a picture, VPU returns from decoding and rolls back its read pointer to the beginning of that picture. In this situation, bit 4 of decodingSuccess is 1. When sequence parameters are changed, bit 20 of decodingSuccess is 1.

`interlacedFrame` where 0 means progressive frame which consists of one frame picture and 1 means interlaced frame which consists of two field picture (top field and bottom field). This variable indicates that the frame is the interlaced frame. If this value is set, the host application may use a de-interlacing filter to enhance image quality.

`mp4PackedPBframe` where 0 means normal frame chunk data and 1 means packed PB frame chunk data. This variable indicates that the frame chunk data is a packed PB frame chunk. If this value is set, the host application must re-use this chunk in the next decoding command. This variable is only valid for MPEG-4 file-play mode.

`h264Npf` indicates that a top or bottom field is absent when NPF occurrs in display picture.

`PictureStructure` is a picture structure in picture coding ext in MP2, interlaced in Video Object Layer in MP4, MBAFF (MB Adaptive frame/field mode) flag in H.264, and FCM in picture header in VC1.

`topFieldFirst` where 0 means bottom field first and 1 means top field first. Ignored if interlacedFrame is 0.

`repeatFirstField` repeats first field for repeat counter.

`progressiveFrame` is a progressive_frame in picture coding extension in MP2.

`vc1_repeatFrame` where 0 means not repeat frame and 1 means repeat frame.

`fieldSequence` is a field sequence in picture extension of MP2.

`decPicHeight` is a picture height of current decoded frame.

`decPicWidth` is a picture width of current decoded frame. For MJPEG decoding, the decPicHeight and decPicWidth are the size of the decoded rotator frame saved in the rotation frame buffer that is registered by the SET_ROTATOR_OUTPUT command. VPU supports the changed resolution decoding. VPU only supports the changed resolution not larger than the original size. For example, the changed sequence of VGA > QVGA > VGA is supported.

`decPicCrop` is a picture crop information of current decoded frame. Only effective with the H.264 decoder.

`aspectRateInfo` H.264 - It is aspect_ratio_idc [7:0] when [31:8] is 0. Otherwise it is ssar_width in [31:16] and sar_height in [15:0]. VC-1 - ASPECT_RATIO h:v are reported in [15:8] : [7:0] as described in the spec. MPEG4 - This value is index of Table 6-12 in ISO/IEC 14496-2; MPEG2 - This value is index of Table 6-3 in ISO/IEC 13818-2.

`frameRateRes` is the numerator part of frame rate fraction. This is the value of time_scale in the H.264 VUI syntax for AVC decoding.

`frameRateDiv` is the denominator part of the frame rate fraction. In case of AVC decoding, this is the value of num_units_in_tick in the H.264 VUI syntax. User can get the frame rate with this parameter. For AVC decoder, frame rate = frameRateRes / (frameRateDiv*2). Otherwise, frame rate = frameRateRes / frameRateDiv.

`vp8ScaleInfo` is VP8 up sampling information. Refer to the Vp8ScaleInfo.

`vp8PicInfo` is VP8 frame header information. Refer to the Vp8PicInfo.

`mvcPicInfo` is MVC related picture information. Refer to MvcPicInfo.

`avcFpaSei` is AVC frame packing arrangement SEI information. Refer to AvcFpaSei.

`frameStartPos` Start position of the frame.

`frameEndPos` End position of the frame.

`consumedByte` Consumed byte in the decoding command.

`mbInfo` is not used in the i.MX 6.

`mvInfo` is not used in the i.MX 6.

`frameBufStat` is not used in the i.MX 6.

`userData` is a motion vector in the decoded picture. If the application does not give the DEC_SET_REPORT_USERDATA command to enable the report before starting one frame decoder, this information is invalid.

4.4.2.31 vpu_versioninfo

```
typedef struct {
                    int fw_major;    /* firmware major version */
                    int fw_minor;    /* firmware minor version */
                    int fw_release;   /* firmware release version */
                    int fw_code;      /* firmware checkin code number */
                    int lib_major;    /* library major version */
                    int lib_minor;    /* library minor version */
                    int lib_release;   /* library release version */
} vpu_versioninfo;
```

Description

This is a data structure to get the VPU firmware and library version:

`fw_major, fw_minor, fw_release` Firmware version, naming convention which are similar to Linux kernel.

`fw_code` is the firmware detail source code commit id.

`lib_major, lib_minor, lib_release` VPU library version, naming convention which are all similar to Linux kernel.

4.4.2.32 VPUMemAlloc

```
typedef struct {
                    int size;
                    unsigned long phy_addr;
                    unsigned long cpu_addr;
                    unsigned long virt_uaddr;
```

```
} vpu_mem_desc;
```

Description

Data structure used when the host application allocates physically contiguous memory for the VPU:

`size` is a requested memory size.

`phy_addr` is a physical base address of the buffer allocated by driver if allocated successfully.

`cpu_addr` is a kernel virtual address corresponding to phy_addr. The programmer of the user-space application does not need to care about this.

`virt_uaddr` is an user-space virtual address corresponding to phy_addr, which the host application can access.

4.4.2.33 iram_t

```
typedef struct iram_t {
                    unsigned long start;
                    unsigned long end;
} iram_t;
```

Description

`start` is a start address of the internal memory for VPU use.

`end` is an end address of internal memory for VPU use.

4.5 API Definitions Overview

This section provides an overview of the VPU API definitions. The basic API architecture is presented together with the operation flow of both decoder and encoder- based VPU API functions.

4.5.1 Basic Architecture

i.MX 6 VPU API has the following three basic categories:

- Control API-API functions for general control of the VPU such as initialization
- Decoder API-API functions for VPU decoding operations
- Encoder API-API functions for VPU encoding operations

i.MX 6 VPU API functions are based on a frame-by-frame picture processing scheme. To run a picture decoder or encoder, the application calls an API function. After completion of the processing, the application can check the results of the picture processing.

To support multi-instance decoding and encoding, i.MX 6 VPU API functions use a handle to specify a certain instance. The handle for each instance is provided when the application creates a new decoder or encoder instance. If the application wants to give a command to a specific instance, the corresponding handle is used in every API function call for that instance.

4.5.1.1 Decoder Operation Flow

To decode a bitstream, the application completes the following steps:

1. Call **vpu_Init**() to initialize the VPU.
2. Open a decoder instance by using **vpu_DecOpen**().

3. To provide the proper amount of bitstream, get the bitstream buffer address by using **vpu_DecGetBitstreamBuffer**().
4. After transferring the decoder input stream, inform the amount of bits transferred into the bitstream buffer by using **vpu_DecUpdateBitstreamBuffer**().
5. Before starting a picture decoder operation, get the crucial parameters for decoder operations such as picture size, frame rate, and required frame buffer size by using **vpu_DecGetInitialInfo**().
6. Using the returned frame buffer requirement, allocate the proper size of the frame buffers, and convey this data to i.MX 6 VPU by using **vpu_DecRegisterFrameBuffer**().
7. Start a picture decoder operation picture-by-picture by using **vpu_DecStartOneFrame**().
8. Wait for the completion of the picture decoder operation interrupt event.
9. Check the results of the decoder operation using **vpu_DecGetOutputInfo**().
10. After displaying n^{th} frame buffer, clear the buffer display flag by using **vpu_DecClrDispFlag**().
11. If there is more bitstream to decode, go to Step 7, otherwise go to the next step.
12. Terminate the sequence operation by closing the instance by using **vpu_DecClose**().
13. Call **vpu_UnInit**() to release the system resources.

The decoder operation flow is shown in the figure below.

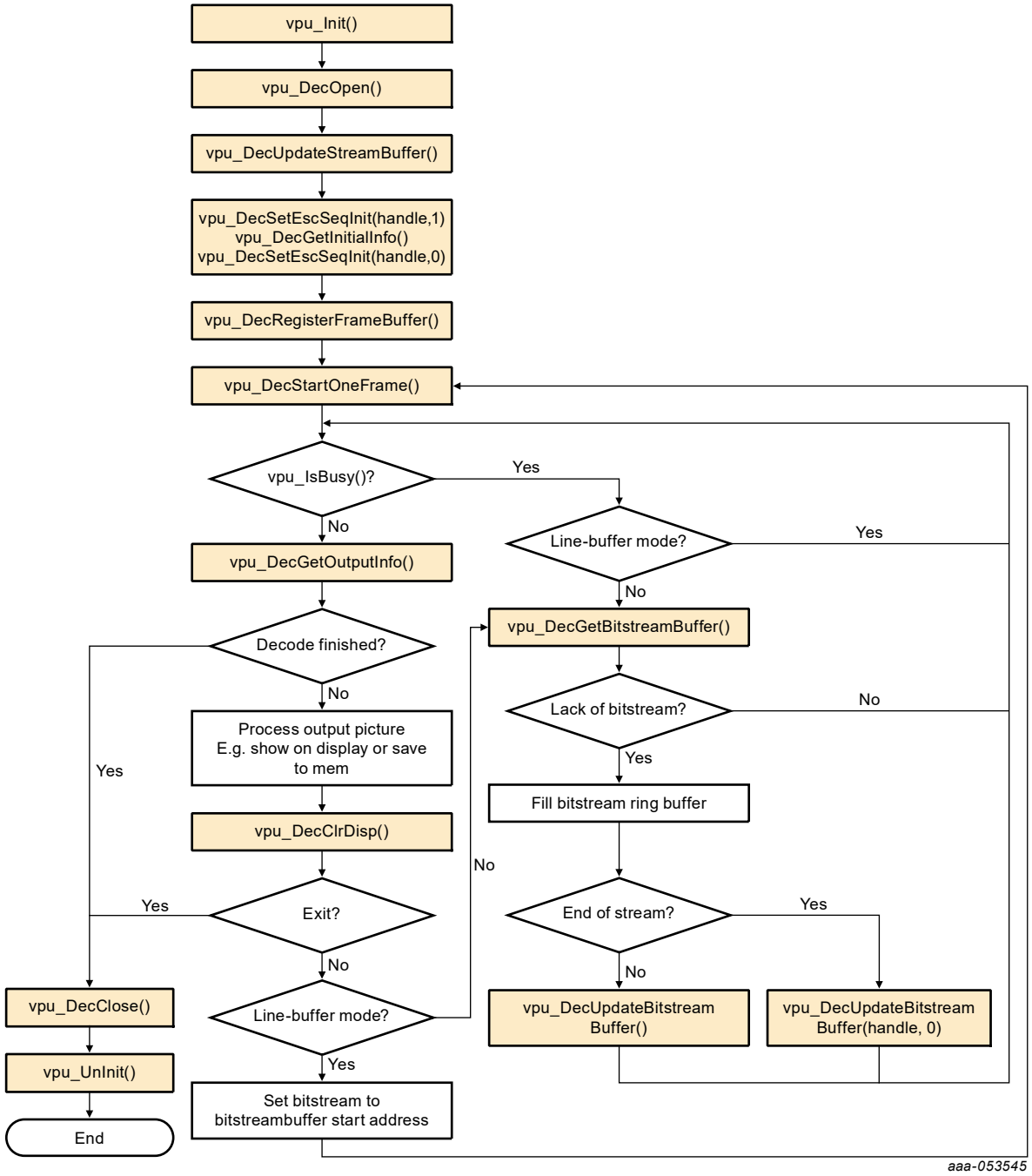

Figure 10. Decoder Operation Flow

4.5.1.2 Encoder Operation Flow

To encode a bitstream, the application completes the following steps:

1. Call **vpu_Init**() to initialize the VPU.
2. Open a encoder instance by using **vpu_EncOpen**().
3. Before starting a picture encoder operation, get crucial parameters for encoder operations such as required frame buffer size by using **vpu_EncGetInitialInfo**().

4. By using the returned frame buffer requirement, allocate size of frame buffers and convey this information to the VPU by using **vpu_EncRegisterFrameBuffer**().
5. Generate high-level header syntax by using **vpu_EncGiveCommand**().
6. Start picture encoder operation picture-by-picture by using **vpu_EncStartOneFrame**().
7. Wait the completion of picture encoder operation interrupt event.
8. After encoding a frame is complete, check the results of encoder operation by using **vpu_EncGetOutputInfo**().
9. If there are more frames to encode, go to Step 4. Otherwise, go to the next step.
10. Terminate the sequence operation by closing the instance using **vpu_EncClose**().
11. Call **vpu_UnInit**() to release the system resources.

The encoder operation flow is shown in the figure below.

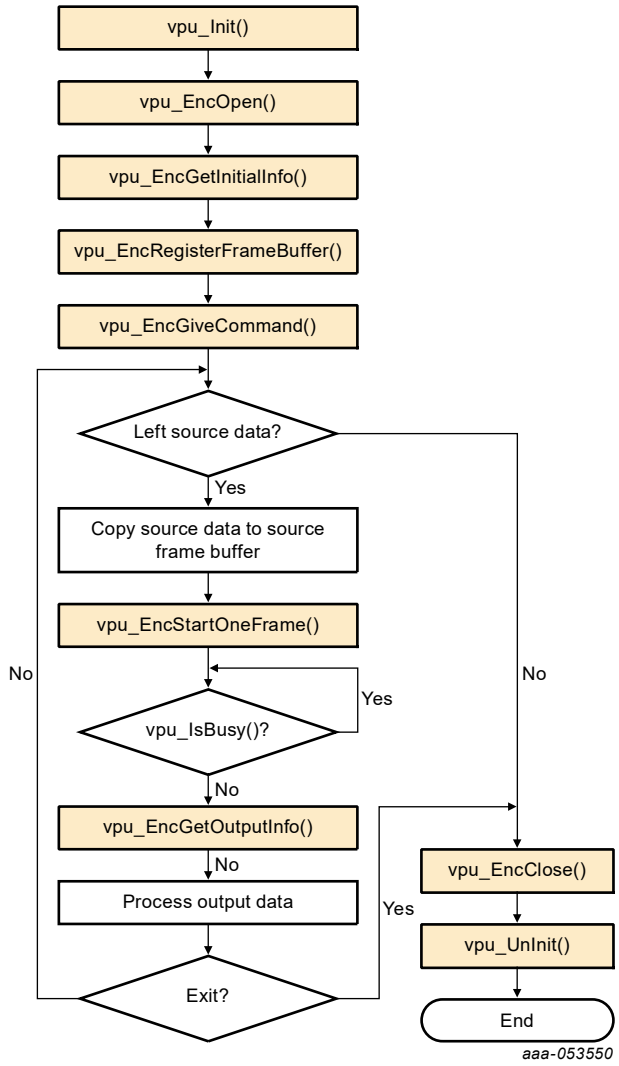

Figure 11. Encoder Operation Flow

4.6 Control API

The following sections describe the control API functions.

4.6.1 vpu_Init()

Prototype

```
RetCode vpu_Init(void *);
```

Parameter

Not used, just defined for extension. The user can set it with null.

Return Value

`RETCODE_SUCCESS` means VPU initialized successfully.

`RETCODE_FAILURE` means VPU initialization unsuccessful.

Description

This function initializes the VPU hardware and proper data structures/resources. The application must call this function before using VPU. If the VPU hardware is initialized after boot at first usage, VPU library does not need to initialize the hardware again. For example, there is no need to load the firmware again. This is transparent to the application.

4.6.2 vpu_UnInit()

Prototype

```
void vpu_UnInit();
```

Parameter

None

Description

This function deinitializes the VPU hardware and releases the resources that are allocated in the vpu_Init() function. The application must call this function before exiting.

4.6.3 vpu_IsBusy()

Prototype

```
int vpu_IsBusy();
```

Parameter

None

Return Value

0 VPU hardware is idle.

1 VPU hardware is busy processing a frame.

Description

This function tells the application if decoder or encoder frame processing is completed or not.

4.6.4 jpu_IsBusy()

Prototype

```
int jpu_IsBusy();
```

Parameter

None

Return Value

0 JPU hardware is idle.

1 JPU hardware is busy processing a frame.

Description

This function tells the application if decoder or encoder frame processing of MJPG format is completed or not. This function is not implemented. Use vpu_IsBusy instead.

4.6.5 vpu_WaitForInt()

Prototype

```
int vpu_WaitForInt(int timeout_in_ms);
```

Parameter

`timeout_in_ms [input]` is wait time in milliseconds.

Return Value

RETCODE_SUCCESS means that the operation is successful.

RETCODE_FAILURE means that the operation failed.

Description

The application waits for the decoder or encoder to complete the interrupt. This function returns immediately if the interrupt has been received. Otherwise, it returns after timeout_in_ms.

4.6.6 vpu_GetVersionInfo()

Prototype

```
RetCode vpu_GetVersionInfo(vpu_versioninfo * verinfo);
```

Parameter

`verinfo [output]` is the pointer to vpu_versionInfo data.

Return Value

RETCODE_SUCCESS means that the version information is acquired successfully.

RETCODE_FAILURE means that the current firmware does not contain any version information.

RETCODE_NOT_INITIALIZED means that VPU is not initialized before calling this function. The application should initialize VPU by calling **vpu_Init**() before calling this function.

Description

This function provides the version information running on the system to the application.

4.6.7 IOGetPhyMem()

Prototype

```
int IOGetPhyMem(vpu_mem_desc * buff);
```

Parameter

`buff [input]` is a pointer to memory information stored in allocated memory. The user needs to input buff > size, then buff >. phy_addr is output after return success.

Return Value

RETCODE_SUCCESS means that the operation is successful.

RETCODE_FAILURE means that the operation failed.

Description

This function allocates physically contiguous memory. When the application calls this function, the driver allocates physically contiguous memory.

4.6.8 IOFreePhyMem()

Prototype

```
int IOFreePhyMem(vpu_mem_desc * buff);
```

Parameter

`buff [input]` is a pointer to memory information stored in allocated memory. The user needs to input buff > size, then buff > phy_addr is output after return success.

Return Value

RETCODE_SUCCESS means that the operation is successful.

RETCODE_FAILURE means that the operation failed.

Description

This function frees the physical memory allocated by IOGetPhyMem back to the system.

4.6.9 IOGetVirtMem()

Prototype

```
int IOGetVirtMem(vpu_mem_desc * buff);
```

Parameter

`buff [input]` is a pointer to memory information stored in allocated memory. The user needs to input buff > size, then buff > phy_addr is output after return success.

Return Value

RETCODE_SUCCESS means that the operation is successful.

RETCODE_FAILURE means that the operation failed.

Description

This function gets the virtual address of the given physical address. If the allocated physical continuous memory needs to be accessed in user space, this function is used to map physical memory.

4.6.10 IOFreeVirtMem()

Prototype

```
int IOFreeVirtMem(vpu_mem_desc * buff);
```

Parameter

`buff [input]` is a pointer to memory information stored in allocated memory. The user needs to input buff > size, then buff > phy_addr is output after return success.

Return Value

RETCODE_SUCCESS means that the operation is successful.

RETCODE_FAILURE means that the operation failed.

Description

This function is used to unmap physical memory to user space.

4.6.11 IOGetIramBase()

Prototype

```
int IOGetIramBase(iram_t * iram);
```

Parameter

`iram [input]` is a pointer to memory information that stores the internal memory.

Return Value

RETCODE_SUCCESS means that the operation is successful.

RETCODE_FAILURE means that the operation failed.

Description

This function is not used in i.MX 6 .

4.6.12 vpu_SWReset()

Prototype

```
RetCode vpu_SWReset(DecHandle handle, int index);
```

Parameter

`handle [input]` is an encoder/decoder handle obtained from vpu_EncOpen()/vpu_DecOpen().

`index [input]` means that the index of instance will be reset.

Return Value

RETCODE_SUCCESS means that the operation is successful.

RETCODE_FAILURE means that the operation failed.

Description

This function resets the instance specified by the *handle* or *index*. Host application can use this function with two methods:

1) Calling with *handle* parameter. If *handle* is given, the *index* parameter will be ignored automatically.

2) Calling with *index* parameter. This method is for special use cases in which the application exists without instance closed, the resources need to be released, and the host knows the exact index of instance.

In normal situations, you should reset VPU with a specified *handle*. You should be confident in what you are doing if resetting VPU with an *index* parameter not a *handle*.

4.7 Encoder API

The following sections describe the encoder API functions.

4.7.1 vpu_EncOpen()

Prototype

```
RetCode vpu_EncOpen(EncHandle * pHandle, EncOpenParam * pop);
```

Parameter

`pHandle [output]` is a pointer to EncHandle type variable which specifies instance for an application. If no instance is available, a null handle is returned.

`pop [input]` is a pointer to a EncOpenParam type structure which describes the parameters for the new encoder instance.

Return Value

RETCODE_SUCCESS means that the new encoder instance opened successfully.

RETCODE_FAILURE means that the new encoder instance not opened successfully. If there is no free instance available, this value is returned in the function call.

RETCODE_INVALID_PARAM means that a given argument parameter, pop, is invalid-it has a null pointer or contains improper values for some member variables.

RETCODE_NOT_INITIALIZED means that VPU is not initialized before calling this function. The application must initialize VPU by calling **vpu_Init**() before calling this function.

Description

To start a new encoder operation, the application must open a new instance. By calling this function, the application gets a handle specifying a new encoder instance. Because i.MX 6 VPU supports multiple instances of codec operations, the application needs this kind of handle for the all running codec instances. Once the application receives a handle, the application uses this handle to represent the target instances for all subsequent encoder-related operations.

4.7.2 vpu_EncClose()

Prototype

```
RetCode vpu_EncClose(EncHandle handle);
```

Parameter

`handle [input]` is an encoder handle obtained from **vpu_EncOpen**().

Return Value

RETCODE_SUCCESS means that the encoder instance closed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be returned if handle has not been obtained by vpu_EncOpen(), for example a decoder handle, or if handle is from an instance which has been closed.

RETCODE_FRAME_NOT_COMPLETE means that the frame decoding or encoding operation is not completed yet and the API function call cannot be performed at 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, the application should call **vpu_EncGetOutputInfo**() or **vpu_DecGetOutputInfo**() to proceed with this function call.

RETCODE_FAILURE_TIMEOUT means that the hardware is already busy with other operation and unavailable for current API calling.

Description

This function is called by the application to close an instance when the application completes the encoding operations and wants to release this instance for other processing. After completion of this function call, the instance referred to by the handle is free. Once the application closes an instance, the application cannot call any further encoder-specific function with this handle before re-opening a new instance with the same handle.

4.7.3 vpu_EncGetInitialInfo()

Prototype

```
RetCode vpu_EncGetInitialInfo(EncHandle handle, EncInitialInfo * info);
```

Parameter

`handle [input]` is an encoder handle obtained from **vpu_EncOpen**().

`info [output]` is a pointer to a EncInitialInfo type structure which describes the parameters required before starting encoder operations.

Return Value

RETCODE_SUCCESS means that receiving the initial parameters completed successfully.

RETCODE_FAILURE means that there is an error getting the configuration information for the encoder.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be returned if handle has not been obtained by **vpu_EncOpen**(), for example a decoder handle, or if handle is of an instance which has been closed.

RETCODE_INVALID_PARAM means that the given argument parameter, info, is invalid. This means that it has a null pointer or contains improper values for some member variables.

RETCODE_CALLED_BEFORE means that the function call is invalid because multiple calls of the current API function for a given instance are not allowed. The encoder initial information has already been received, so this function call is meaningless and not allowed.

RETCODE_FAILURE_TIMEOUT means that the hardware is already busy with other operation and unavailable for current API calling.

Description

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

4.7.4 vpu_EncGetBitstreamBuffer()

Prototype

```
RetCode vpu_EncGetBitstreamBuffer(EncHandle handle,
                  PhysicalAddress * prdPrt,
                  PhysicalAddress * pwrPtr, Uint32 * size);
```

Parameter

`handle [input]` is an encoder handle obtained from **vpu_EncOpen**().

`prdPrt [output]` is a stream buffer read pointer for the current encoder instance.

`pwrPtr [output]` is a stream buffer write pointer for the current encoder instance.

`size [output]` is a variable specifying the available space in the bitstream buffer for the current encoder instance.

Return Value

RETCODE_SUCCESS means that the required information for encoder stream buffer is received successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be returned if handle has not been obtained by **vpu_EncOpen**(), for example a decoder handle, or if handle is of an instance which has been closed.

RETCODE_INVALID_PARAM means that given argument parameters, prdPrt, pwrPtr, or size, are invalid. This means that they have a null pointer or contain improper values for some member variables.

Description

After encoding a frame, the application must get the bitstream from the encoder by using the stream location and the maximum size. The application gets the information by calling this function.

4.7.5 vpu_EncUpdateBitstreamBuffer()

Prototype

```
RetCode vpu_EncUpdateBitstreamBuffer(EncHandle handle, Uint32 size);
```

Parameter

handle [input] Encoder handle obtained from vpu_EncOpen()

size [input] Variable specifying the amount of bits retrieved from the bitstream buffer for the current encoder instance

Return Value

RETCODE_SUCCESS: Putting new stream data completed successfully

RETCODE_INVALID_HANDLE: Given handle for current API function call, handle, is invalid. This return code might be returned if handle has not been obtained by **vpu_EncOpen**(), for example a decoder handle, or if handle is of an instance which has been closed.

RETCODE_INVALID_PARAM: Given argument parameter, size, is invalid-it is larger than the value obtained from **vpu _EncGetBitstreamBuffer** ()

Description

The application must let the encoder know how much bitstream has been transferred from the address obtained from **vpu _EncGetBitstreamBuffer** (). By giving the size as an argument, the API automatically handles pointer wrap-around and updates the read pointer.

4.7.6 vpu_EncRegisterFrameBuffer()

Prototype

```
RetCode vpu_EncRegisterFrameBuffer(EncHandle handle,
          FrameBuffer * bufArray, int num, int frameBufStride, int
 sourceBufStride,
          PhysicalAddress subSampBaseA,PhysicalAddress subSampBaseB,
 EncExtBufInfo *pBufInfo);
```

Parameter

`handle [input]` is an encoder handle obtained from **vpu_EncOpen**().

`bufArray [input]` is a pointer to the first element of an array of FrameBuffer data structure.

`num [input]` is a number of frame buffers.

`frameBufStride [input]` is a stride value of the given frame buffers for encoder.

`sourceBufStride [input]` is a stride value of the source frame buffer for encoder.

`subSampBaseA [input]` is a buffer address for saving a sub-sampled image.

`subSampBaseB [input]` is a buffer address for saving a sub-sampled image.

`pBufInfo [input]` is a buffer address for saving extension buffer info. See EncExtBufInfo for details.

The distance between a pixel in a row and the corresponding pixel in the next row is called a stride. The value of a 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, a stride can have values greater than the picture width in pixels.

The application should not set a stride value smaller than the picture width. For the Y component, the application must allocate at least a space of size (frame height x stride), and for Cb or Cr components, (frame height/2 x stride/2).

For MJPEG encoding, the address of the frame buffer is not necessary. Only the frameBufStride and frameBufStride values are necessary.

Return Value

RETCODE_SUCCESS means that registering the frame buffers completed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be returned if handle has not been obtained by **vpu_EncOpen**(), for example a decoder handle, or if handle is of an instance which has been closed.

RETCODE_WRONG_CALL_SEQUENCE means that the current API function call is invalid considering the allowed sequences between API functions. In this situation, the application may have called this function before successfully calling **vpu_EncGetInitialInfo**(). This function should be called after successfully calling **vpu_EncGetInitialInfo**().

RETCODE_INVALID_FRAME_BUFFER means that the argument bufArray is invalid or not initialized.

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

RETCODE_INVALID_STRIDE means that the given argument stride is invalid. This means that it is 0, or is not a multiple of 8.

RETCODE_CALLED_BEFORE means that the function call is invalid because multiple calls of the current API function for a given instance are not allowed. The encoder initial information has already been received, so this function call is meaningless and not allowed.

Description

This function registers frame buffers requested by **vpu_EncGetInitialInfo**(). The frame buffers pointed to by bufArray are managed internally within VPU. These include reference frames, reconstructed frames, and so on. The application must not change the contents of the array of frame buffers during the life time of the instance. num must not be less than minFrameBufferCount obtained by **vpu_EncGetInitialInfo**().

4.7.7 vpu_EncStartOneFrame()

Prototype

```
RetCode vpu_EncStartOneFrame(EncHandle handle, EncParam * param);
```

Parameter

`handle [input]` is an encoder handle obtained from vpu_EncOpen().

`param [input]` is a pointer to a EncParam type structure which describes the picture encoding parameters for the current encoder instance.

Return Value

RETCODE_SUCCESS means that encoding a new frame started successfully. This return value does not mean that encoding a frame completed successfully.

RETCODE_FAILURE means that there is an error in starting one frame encoding operation.

RETCODE_INVALID_HANDLE means that a given handle for current API function call is invalid. This code might be returned if handle has not been obtained by **vpu_EncOpen**(), for example a decoder handle, or if handle is of an instance which has been closed.

RETCODE_WRONG_CALL_SEQUENCE means that the current API function call is invalid considering the allowed sequences between API functions. In this situation, the application may have called this function before successfully calling **vpu_EncRegisterFrameBuffer**(). This function should be called after successfully calling **vpu_EncRegisterFrameBuffer**().

RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid. This means that it has a null pointer, or contains improper values for some member variables.

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

RETCODE_FAILURE_TIMEOUT means that the hardware is already busy with other operation and unavailable for current API calling.

Description

This function starts by encoding one frame. Returning from this function does not mean the completion of encoding one frame, only that encoding of one frame successfully initiated. This function should be followed by **vpu_EncGetOutputInfo**() with the same encoder handle. Before **vpu_EncGetOutputInfo**() is called,

the application can't call other API functions except for **vpu_IsBusy**(), **vpu_EncGetBitstreamBuffer**(), or **vpu_EncUpdateBitstreamBuffer**().

4.7.8 vpu_EncGetOutputInfo()

Prototype

```
RetCode vpu_EncGetOutputInfo(EncHandle handle, EncOutputInfo * info)
```

Parameter

`handle [input]` is an encoder handle obtained from **vpu_EncOpen**().

`info [output]` is a pointer to an EncOutputInfo type structure which describes picture encoding results for the current encoder instance.

Return Value

RETCODE_SUCCESS means that the output information of current frame encoding received successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be returned if handle has not been obtained by **vpu_EncOpen**(), for example a decoder handle, or if handle is of an instance which has been closed.

RETCODE_WRONG_CALL_SEQUENCE means that the current API function call is invalid considering the allowed sequences between API functions. In this situation, the application may have called this function before successfully calling **vpu_EncStartOneFrame**(). This function should be called after successfully calling **vpu_EncStartOneFrame**().

RETCODE_INVALID_PARAM means that the given argument parameter, info, is invalid. This means that it has a null pointer, or contains improper values for some member variables.

Description

This function gives the information about the encoding output such as the picture type, the address and size of the generated bitstream, the number of generated slices, the end addresses of the slices, and the macroblock bit position information. The host application should call this function after frame encoding is complete and before starting further processing.

4.7.9 vpu_EncGiveCommand()

Prototype

```
RetCode vpu_EncGiveCommand(EncHandle handle, CodecCommand cmd, void *param);
```

Parameter

`handle [input]` is an encoder handle obtained from **vpu_EncOpen**().

`cmd [input]` is a variable specifying the command of CodecComand type.

`param [intput/output]` is a pointer to a command-specific data structure which describes picture I/O parameters for the current encoder instance.

Return Value

RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be returned if handle has not been obtained by **vpu_EncOpen**(), for example a decoder handle, or if handle is of an instance which has been closed.

RETCODE_FRAME_NOT_COMPLETE means that frame encoding operation is not complete, 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, the application should call **vpu_EncGetOutputInfo**() or **vpu_DecGetOutputInfo**() to proceed with this function call.

Description

This function is provided to give the application a certain level of freedom for reconfiguring the encoder operation after creating an encoder instance. The options which can be changed dynamically while encoding a video sequence as well as some command-specific return codes are shown in table below.

Table 3. Encoder Commands

Command	Description
ENABLE_ROTATION	handle is ignored. This command returns RETCODE_SUCCESS.
DISABLE_ROTATION	handle is ignored. This command returns RETCODE_SUCCESS.
ENABLE_MIRRORING	handle is ignored. This command returns RETCODE_SUCCESS.
DISABLE_MIRRORING	handle is ignored. This command returns RETCODE_SUCCESS.
SET_MIRROR_DIRECTION	handle is a pointer to MirrorDirection. *param should be one of the following: • MIRDIR_NONE means no mirroring. • MIRDIR_VER means vertical mirroring. • MIRDIR_HOR means horizontal mirroring. • MIRDIR_HOR_VER means both directions mirroring. Return values are as follows: RETCODE_SUCCESS means that the given mirroring direction is valid. RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given mirroring direction is invalid.
SET_ROTATION_ANGLE	param is a pointer to an integer which represents rotation angle in degrees. Rotation angle should be 0, 90, 180, or 270. Return values are as follows: RETCODE_SUCCESS means that the given rotation angle is valid. RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given rotation angle is invalid. ***Note:*** *Rotation angle cannot be changed after sequence initialization since it might cause problems in handling frame buffers.*
ENC_GET_SPS_RBSP	param is a pointer to an EncParamSet type structure. The first variable, paraSet, is a physical address where the generated stream is located. Size is the size of the stream in bytes. Return values are as follows: RETCODE_SUCCESS SPS means that it is successfully generated and available at the received buffer pointer. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, current instance might not be an AVC (H.264) encoder instance. RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer or contains improper values for some member variables.
ENC_GET_PPS_RBSP	param is a pointer to an EncParamSet type structure. Return values are as follows: RETCODE_SUCCESS PPS means that it is successfully generated and available at the received buffer pointer.

Table 3. Encoder Commands*...continued*

Command	Description
	RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, current instance might not be an AVC (H.264) encoder instance. RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer or contains improper values for some member variables.
ENC_PUT_MP4_HEADER	param is a pointer to an EncHeaderParam structure, where buf is a physical address pointing to the generated stream location, and size is the size of the generated stream in bytes. headerType is a type of header that the application wants to generate and has values such as VOL_HEADER, VOS_HEADER, or VO_HEADER. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax is successfully inserted. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, current instance might not be an MPEG-4 encoder instance. RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer or contains improper values for some member variables.
ENC_PUT_AVC_HEADER	param is a pointer to an EncHeaderParam structure, where buf is a physical address pointing the generated stream location and size is the size of generated stream in bytes. headerType is a type of header that the application wants to generate and has values such as SPS_RBSP or PPS_RBSP. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax successfully inserted. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, current instance might not be an AVC (H.264) encoder instance. RETCODE_INVALID_PARAM means that the given argument, param or headerType, is invalid. It has a null pointer or contains improper values for some member variables
ENC_SET_SEARCHRAM_ PARAM	The command is not used in i.MX 6 .
ENC_SET_INTRA_MB_ REFRESH_NUMBER	param is a pointer to an integer which represents the intra refresh number. The intra refresh number should be between 0 and the macroblock number of the encoded picture. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax is successfully inserted,
ENC_ENABLE_HEC	param is ignored. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax is successfully inserted. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, current instance might not be an MPEG-4 encoder instance.
ENC_DISABLE_HEC	param is ignored. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax successfully inserted. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, current instance might not be an MPEG-4 encoder instance.
ENC_SET_SLICE_INFO	param is a pointer to an EncSliceMode structure, where sliceMode enables a multi slice structure. sliceSizeMode represents the mode of calculating one slicesize. sliceSize is the size of one slice. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax is successfully inserted. RETCODE_INVALID_PARAM means that the given argument parameter, param or header Type, is invalid. It has a null pointer or contains improper values for some member variables.
ENC_SET_GOP_NUMBER	param is a pointer to an integer which represents the GOP number. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax is successfully inserted.

Table 3. Encoder Commands*...continued*

Command	Description
	RETCODE_INVALID_PARAM means that the given argument parameter, param or header Type, is invalid. It has a null pointer or contains improper values for some member variables.
ENC_SET_INTRA_QP	param is a pointer to an integer which represents constant I frame QP. Constant I frame QP should be between 1 and 31 for MPEG-4, and between 0 and 51 for AVC (H.264). Return values are as follows: RETCODE_SUCCESS means that the requested header syntax is successfully inserted. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, current instance might not be an encoder instance. RETCODE_INVALID_PARAM means that the given argument parameter, param or header Type, is invalid. It has a null pointer or contains improper values for some member variables.
ENC_SET_BITRATE	param is a pointer to an integer which represents the bitrate. The bitrate should be between 0 and 32767. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax is successfully inserted. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, current instance might not be an encoder instance. RETCODE_INVALID_PARAM means that the given argument parameter, param or header Type, is invalid. It has a null pointer or contains improper values for some member variables.
ENC_SET_FRAME_RATE	param is a pointer to an integer which represents the frame rate value. The frame rate should be greater than 0. Return values are as follows: RETCODE_SUCCESS means that the requested header syntax inserted successfully. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance. In this instance, the current instance might not be an encoder instance. RETCODE_INVALID_PARAM means that the given argument parameter, param or header Type, is invalid. It has a null pointer or contains improper values for some member variables.
ENC_SET_REPORT_ MBINFO	Not used in i.MX 6 .
ENC_SET_REPORT_ MVINFO	Not used in i.MX 6 .
ENC_SET_REPORT_ SLI CEINFO	param is a pointer to an EncReportInfo. addr cannot be a null pointer when the enable flag is 1, so the user needs to allocate memory according to mvInfoBufSize returned by **vpu_EncGetInitialInfo**(). The user can call malloc() to allocate the buffer since continuous physical memory is not needed. Return values are as follows: RETCODE_INVALID_PARAM means that the given argument parameter, param is invalid. It has a null pointer or addr in EncReportInfo is a null pointer when enable is 1.
ENC_SET_INTRA_ REFRESH_MODE	Set intra refresh mode. It must be called before vpu_EncGetInitialInfo takes effect. • 0 - random intra refresh mode • 1 - consecutive intra refresh mode
ENC_ENABLE_SOF_STUFF	Pad stuffing zero bytes to the end of JPEG SOF fields. • 0 - disable stuffing • 1 - enable stuffing

4.8 Decoder API

The following sections describe the decoder API functions.

4.8.1 vpu_DecOpen()

Prototype

```
RetCode vpu_DecOpen(DecHandle * pHandle, DecOpenParam * pop);
```

Parameter

`pHandle [output]` is a pointer to a DecHandle type variable which specifies each instance for an application.

`pop [input]` is a pointer to a DecOpenParam type structure which describes the required parameters for creating a new decoder instance.

Return values:

RETCODE_SUCCESS means that the new decoder instance created successfully.

RETCODE_FAILURE means that the new decoder instance did not open successfully. If there is no free instance available, this value is returned in the function call.

RETCODE_INVALID_PARAM means that the given argument parameter, pop, is invalid. It has a null pointer or contains improper values for some member variables.

RETCODE_NOT_INITIALIZED means that VPU is not initialized before calling this function. The application must initialize the VPU by calling **vpu_Init**() before calling this function.

Description

To decode, the application must open the decoder. By calling this function, the application receives a handle by which the application can refer to a decoder instance. Since VPU is a multiple instance codec, the application requires this kind of handle. Once the application receives a handle, the application must pass the handle to all subsequent decoder-related functions.

4.8.2 vpu_DecClose()

Prototype

```
RetCode vpu_DecClose(DecHandle handle);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

Return Value

RETCODE_SUCCESS means that the current decoder instance closed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This return code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed.

RETCODE_FAILURE_TIMEOUT means that VPU is busy with another task, or there is something wrong with VPU. In normal operation, the API call should not return a RETCODE_FAILURE_TIMEOUT value. If the application receives this value, VPU internal function may be corrupted.

RETCODE_FAILURE_TIMEOUT means that hardware is already busy with other operation and unavailable for current API calling.

Description

When the application is finished decoding a sequence and wants to release this instance for other processing, the application should close the instance. After completion of this function call, the instance referred to by

handle is free. Once the application closes an instance, the application cannot call any further decoder-specific function with this handle before re-opening a new decoder instance with the same handle.

4.8.3 vpu_DecGetInitialInfo()

Prototype

```
RetCode vpu_DecGetInitialInfo(DecHandle handle, DecInitialInfo * info);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

`info [output]` is a pointer to a DecInitialInfo data structure.

Return Value

RETCODE_SUCCESS means that the required information of the stream data to be decoded is received successfully.

RETCODE_FAILURE means that there is an error in getting the configuration information for the decoder.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed.

RETCODE_INVALID_PARAM means that the given argument parameter, info, is invalid. It has a null pointer or contains improper values for some member variables.

RETCODE_FAILURE_TIMEOUT means that VPU is busy with another task, or there is something wrong with the VPU. In normal operation, the API call should not return a RETCODE_FAILURE_TIMEOUT value. If the application receives this value, the VPU internal function may be corrupted.

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

RETCODE_CALLED_BEFORE means that the function call is invalid because multiple calls of the current API function for a given instance are not allowed. The decoder initial information has been already received, so this function call is meaningless and not allowed.

RETCODE_FAILURE_TIMEOUT means that the hardware is already busy with other operation and unavailable for current API calling.

Description

The application must pass the address of a DecInitialInfo structure, where the decoder stores the information such as picture size, number of necessary frame buffers, and so on. For details, see the definition of the DecInitialInfo data structure in Section 4.4.2.21. This function should be called after creating a decoder instance and before starting frame decoding. The application must provide sufficient amount of bitstream to the decoder by calling **vpu_DecUpdateBitstreamBuffer**() to avoid bitstream buffer emptying before this function returns.

If the application cannot ensure to feed enough data for the stream, the application can use the forced escape option using **vpu_DecSetEscSeqInit**().

4.8.4 vpu_DecSetEscSeqInit()

Prototype

```
RetCode vpu_DecSetEscSeqInit(DecHandle handle, int escape);
```

Parameter

`handle [input]` is a decoder handle obtained from vpu_DecOpen().

`escape [input]` is a flag to enable or disable forced escape from SEQ_INIT.

Return Value

RETCODE_SUCCESS means that the force escape flag successfully provided to the BIT processor.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This return code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed.

Description

This is a special function to provide a way of escaping the VPU hanging during DEQ_SEQ_INIT. When this flag is set to 1 and the stream buffer becomes empty, the VPU automatically terminates the DEC_SEQ_INIT operation. If the target application ensures that a high layer header syntax is periodically sent through the channel, the application does not need this option. However, if the target application cannot ensure that a high layer header syntax is periodically sent through the channel (such as file-play mode). This function is useful to avoid VPU hanging because of crucial errors in the header syntax.

Note: *This flag is applied to all decoder instances together. Therefore, it is recommended to reset this flag to 0 after successfully finishing the sequence initialization.*

4.8.5 vpu_DecGetBitstreamBuffer()

Prototype

```
RetCode vpu_DecGetBitstreamBuffer(DecHandle handle,
                  PhysicalAddress * paRdPtr,
                  PhysicalAddress * paWrPtr, Uint32 * size);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

`paRdPtr [output]` is a stream buffer read pointer for the current decoder instance.

`paWrPtr [output]` is a stream buffer write pointer for the current decoder instance.

`size [output]` is a variable specifying the available space in the bitstream buffer for the current decoder instance.

Return Value

RETCODE_SUCCESS means that the required information for the decoder stream buffer received successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This return code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed.

RETCODE_INVALID_PARAM means that the given argument parameter, paRdPtr, paWrPtr, or size, is invalid. It has a null pointer or given values for some member variables have improper values.

Description

Before decoding a bitstream, the application must give the bitstream data to the decoder. First, the application must know where bitstream can be placed and the maximum size. The application receives this information from this function. For VPU, using the data from this function is more efficient than providing an arbitrary bitstream buffer to the decoder.

Note: *The given size is the total sum of the free space in the ring buffer. Therefore, when the application downloads a bitstream of this given size, Wrptr can reach the end of the stream buffer. In this case, the application should wrap-around Wrptr to the beginning of the stream buffer and download the remaining bits. If not, the decoder operation can fail.*

4.8.6 vpu_DecUpdateBitstreamBuffer()

Prototype

```
RetCode vpu_DecUpdateBitstreamBuffer(DecHandle handle, Uint32 size);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

`size [input]` is a variable specifying the amount of bits transferred into the bitstream buffer for the current decoder instance.

Return Value

RETCODE_SUCCESS means that putting new stream data completed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be returned if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed.

RETCODE_INVALID_PARAM means that the given argument parameter, size, is invalid. This means that it is larger than the value obtained from **vpu_DecGetBitstreamBuffer**(), or larger than the available space in the bitstream buffer.

RETCODE_FAILURE_TIMEOUT means that VPU is busy with another task, or there is something wrong with VPU. In normal operation, the API call should not return a RETCODE_FAILURE_TIMEOUT value. If the application receives this value, the VPU internal function may be corrupted.

Description

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

4.8.7 vpu_DecRegisterFrameBuffer()

Prototype

```
RetCode vpu_DecRegisterFrameBuffer(DecHandle handle,
                    FrameBuffer * bufArray, int num, int stride,
                    DecBufInfo * pBufInfo);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

`bufArray [input]` is a pointer to the first element of an array of FrameBuffer for the current decoder instance.

`num [input]` is a number of frame buffers.

`stride [input]` is a stride value of the given frame buffers.

`pBufInfo [input]` is a pointer to a DecBufInfo type structure which describes the additional work buffers. Only sliceSaveBuffer is declared by this structure.

Return Value

RETCODE_SUCCESS means that registering the frame buffer information completed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This return code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed.

RETCODE_FAILURE_TIMEOUT means that VPU is busy with another task, or there is something wrong with VPU. In normal operation, the API call should not return a RETCODE_FAILURE_TIMEOUT value. If the application receives this value, the VPU internal function may be corrupted.

RETCODE_WRONG_CALL_SEQUENCE means that the current API function call is invalid considering the allowed sequence between API functions. In this case, the application might have called this function before successfully calling **vpu_DecGetInitialInfo**().

RETCODE_INVALID_FRAME_BUFFER means that bufArray is invalid. It is not initialized, or is not valid anymore.

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

RETCODE_INVALID_STRIDE means that the given argument stride is invalid. It is smaller than the decoded picture width, or is not a multiple of 8.

RETCODE_CALLED_BEFORE means that the function call is invalid because multiple calls of the current API function for a given instance are not allowed. The decoder initial information has been already received, so this function call is meaningless and not allowed.

Description

This function is used for registering frame buffers with the information from **vpu_DecGetInitialInfo**(). The frame buffers pointed to by bufArray are managed internally within VPU. These include reference frames, reconstructed frame, and so on. The 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 from **vpu_DecGetInitialInfo**().

4.8.8 vpu_DecStartOneFrame()

Prototype

```
RetCode vpu_DecStartOneFrame(DecHandle handle, DecParam * param);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

`param [input]` is a pointer to a DecParam type structure which describes the decoder options.

Return value

RETCODE_SUCCESS means that decoding a new frame started successfully. This return value does not mean that decoding a frame completed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed.

RETCODE_WRONG_CALL_SEQUENCE means that the current API function call is invalid considering the allowed sequence between API functions. The application might have called this function before successfully calling **vpu_DecRegisterFrameBuffer**(). This function should be called after successfully calling **vpu_DecRegisterFrameBuffer**().

RETCODE_DEBLOCKING_OUTPUT_NOT_SET means that the de-blocking filter option is activated but required de-blocking output information is not available. If de-blocking filter is enabled for MPEG-4, the application should register the frame buffer information of de-blocking filtered output using **vpu_DecGiveCommand**().

RETCODE_FAILURE_TIMEOUT means that hardware is already busy with other operation and unavailable for current API calling.

Description

This function starts by decoding one frame. Returning from this function does not mean the completion of decoding one frame, only that encoding of one frame successfully initiated. If this event is signaled, then **vpu_DecGetOutputInfo**() is called to get the decoded output information. Every call of this function should be matched with **vpu_DecGetOutputInfo**() with the same handle. Before **vpu_DecGetOutputInfo**() is called, the application cannot call another API function except for **vpu_IsBusy**(), **vpu_DecGetBitstreamBuffer**(), or **vpu_DecUpdateBitstreamBuffer**().

When the application uses pre-scan mode, there is only a very small chance that the decoder may hang. For the VC-1 SP/MP decoder, pre-scan mode is not supported.

4.8.9 vpu_DecGetOutputInfo()

Prototype

```
RetCode vpu_DecGetOutputInfo(DecHandle handle, DecOutputInfo * info);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

`info [output]` is a pointer to a DecOutputInfo type structure which describes the picture decoding results for the current decoder instance.

Return Value

RETCODE_SUCCESS means that receiving the output information of current frame completed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This return code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed. Also, this value is returned when **vpu_DecStartOneFrame**() is matched with **vpu_DecGetOutputInfo**() with different handles.

RETCODE_WRONG_CALL_SEQUENCE means that the current API function call is invalid considering the allowed sequence between API functions. **vpu_DecStartOneFrame**() with the same handle might not have been called before calling this function

RETCODE_INVALID_PARAM means that the given argument parameter, pInfo, is invalid. It has a null pointer or contains improper values for some member variables.

Description

The application received the output information of the decoder by calling this function after the VPU_INT_PIC_RUN_NAME event is signaled. The output information includes the frame buffer information containing the reconstructed image. The host application calls this function after the frame decoding is finished and before starting further processing.

Note: *If pre-scan mode is enabled, the application should check prescanResult. If the value of prescanResult = 0, the other output information is meaningless.* ***vpu_DecStartOneFrame****() and* ***vpu_DecGetOutputInfo****() must be matched.*

4.8.10 vpu_DecBitBufferFlush()

Prototype

```
RetCode vpu_DecBitBufferFlush(DecHandle handle);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

Return Value

RETCODE_SUCCESS means that receiving the output information of the current frame completed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This return code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed. Also, this value is returned when **vpu_DecStartOneFrame**() is matched with **vpu_DecGetOutputInfo**() with different handles.

RETCODE_WRONG_CALL_SEQUENCE means that the current API function call is invalid considering the allowed sequence between API functions. **vpu_DecRegisterFrameBuffer**() with the same handle might not have been called before calling this function.

Description

The application flushes the bitstream in the decoder bitstream buffer without decoding by calling this function. If the bitstream buffer is flushed, the read and write pointers of the bitstream buffer of each instance are set to the bitstream buffer start address.

4.8.11 vpu_DecClrDispFlag()

Prototype

```
RetCode vpu_DecClrDispFlag(DecHandle handle, int index);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

`index [input]` is a frame buffer index to be cleared.

Return Value

RETCODE_SUCCESS means that receiving the output information of the current frame completed successfully.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This return code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed. Also, this value is returned when **vpu_DecStartOneFrame**() is matched with **vpu_DecGetOutputInfo**() with different handles.

RETCODE_WRONG_CALL_SEQUENCE means that the current API function call is invalid considering the allowed sequence between API functions. **vpu_DecRegisterFrameBuffer**() with the same handle might not have been called before calling this function.

RETCODE_INVALID_PARAM means that the given argument parameter, index, is invalid. It has improper values.

Description

The application clears the display flag of each frame buffer by calling this function after creating a decoder instance. If the display flag of the frame buffer is cleared, the frame buffer can be used in the decoding process. Therefore, the application controls displaying a buffer by clearing the display flag which is set by VPU at every display index output process. This API is not needed for the STD_MJPG codec.

4.8.12 vpu_DecGiveCommand()

Prototype

```
RetCode vpu_DecGiveCommand(DecHandle handle, CodecCommand cmd, void *param);
```

Parameter

`handle [input]` is a decoder handle obtained from **vpu_DecOpen**().

`cmd [input]` is a variable specifying the given command of CodecComand type.

`param [input/output]` is a pointer to a command-specific data structure which describes picture I/O parameters for the current decoder instance.

Return Value

RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined or not allowed in the current instance.

RETCODE_INVALID_HANDLE means that the given handle for current API function call is invalid. This return code might be caused if handle has not been obtained by **vpu_DecOpen**(), or if handle is of an instance which has been closed.

RETCODE_FAILURE_TIMEOUT means that hardware is already busy with other operation and unavailable for current API calling.

Description

This function is provided to give applications a certain level of freedom for reconfiguring decoder operations after creating a decoder instance. The options which can be changed dynamically while decoding a video sequence are shown in the table below.

Table 4. Decoder Commands

Command	Description
ENABLE_ROTATION	Enables rotation of the post-rotator. param is ignored. Returns RETCODE_SUCCESS.
DISABLE_ROTATION	Disables rotation of the post-rotator. param is ignored. Returns RETCODE_SUCCESS.
ENABLE_MIRRORING	Enables mirroring of the post-rotator. param is ignored. Returns RETCODE_SUCCESS.
DISABLE_MIRRORING	Disables mirroring of the post-rotator. param is ignored. Returns RETCODE_SUCCESS.
SET_MIRROR_ DIRECTION	Sets the mirror direction of the post-rotator. param is a pointer to MirrorDirection. *param should be one of the following: • MIRDIR_NONE-No mirroring • MIRDIR_VER-Vertical mirroring

Table 4. Decoder Commands*...continued*

Command	Description
	• MIRDIR_HOR-Horizontal mirroring • MIRDIR_HOR_VER-Both directions Return values are as follows: RETCODE_SUCCESS means that the given mirroring direction is valid. RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given mirroring direction is invalid.
SET_ROTATION_ANGLE	Sets the counter-clockwise angle for post-rotation. param a pointer to an integer which represents rotation angle in degrees. The rotation angle should be 0, 90, 180, or 270. Return values are as follows: RETCODE_SUCCESS means that the given rotation angle is valid. RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given rotation angle is invalid.
SET_ROTATOR_OUTPUT	Sets the rotator output buffer address. param a pointer to a structure representing the physical addresses of the YCbCr components of the output frame. For storing the rotated output for a display, at least one more frame buffer should be allocated. When multiple display buffers are required, the application changes the buffer pointer of the rotated output at every frame by issuing this command. Return values are as follows: RETCODE_SUCCESS means that the given frame buffer pointer is valid. RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given frame buffer pointer is invalid.
SET_ROTATOR_STRIDE	Sets the stride size of the frame buffer containing rotated output. param is the stride value of the rotated output. Return values are as follows: RETCODE_SUCCESS means that the given stride value is valid. RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given stride value is invalid. The stride value must be greater than 0 and a multiple of 8.
DEC_SET_SPS_RBSP	Applies the SPS stream to the decoder received from a certain out-of-band reception scheme. The stream should be in RBSP format and big endian. param is a pointer to a Dec ParamSet structure. 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 means that transferring a SPS RBSP to a decoder completed successfully. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is undefined, or not allowed in the current instance. In this case, the current instance might not be an AVC (H.264) decoder instance. RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer or contains improper values for some member variables.
DEC_SET_PPS_RBSP	Applies the PPS stream to the decoder received from a certain out-of-band reception scheme. The stream should be in RBSP format and big endian. param is a pointer to a Dec ParamSet structure. 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 means that transferring a PPS RBSP to decoder completed successfully. RETCODE_INVALID_COMMAND means that the given argument, cmd, is invalid. It is 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 means that the given argument, param, is invalid. It has a null pointer, or contains improper values for some member variables.
ENABLE_DERING	Enables VPU internal dering operation. Returns RETCODE_SUCCESS.
DISABLE_DERING	Disables VPU internal dering function. Returns RETCODE_SUCCESS.

Table 4. Decoder Commands*...continued*

Command	Description
DEC_SET_REPORT_ BUFSTAT	param is a pointer to an DecReportInfo. addr cannot be a null pointer when the enable flag is 1, so the user needs to allocate memory according to frameBufStatBufSize returned by **vpu_DecGetInitialInfo**(). The user can call malloc() to allocate the buffer since continuous physical memory is not needed. Return values are as follows: RETCODE_INVALID_PARAM means that the given argument parameter, param is invalid. It has a null pointer, or addr in EncReportInfo is a null pointer when enable is 1.
DEC_SET_REPORT_ MBINFO	param is a pointer to an DecReportInfo. addr cannot be a null pointer when the enable flag is 1, so the user needs to allocate memory according to frameBufStatBufSize returned by **vpu_DecGetInitialInfo**(). The user can call malloc() to allocate the buffer since continuous physical memory is not needed. Return values are as follows: RETCODE_INVALID_PARAM means that the given argument parameter, param is invalid. It has a null pointer, or addr in EncReportInfo is a null pointer when enable is 1.
DEC_SET_REPORT_ MVINFO	param is a pointer to an DecReportInfo. addr cannot be a null pointer when the enable flag is 1, so the user needs to allocate memory according to frameBufStatBufSize returned by **vpu_DecGetInitialInfo**(). The user can call malloc() to allocate the buffer since continuous physical memory is not needed. Return values are as follows: RETCODE_INVALID_PARAM means that the given argument parameter, param is invalid. It has a null pointer, or addr in EncReportInfo is a null pointer when enable is 1.
DEC_SET_REPORT_ USERDATA	param is a pointer to an DecReportInfo. addr cannot be a null pointer and size cannot be zero when the enable flag is 1, so the user needs to allocate memory. The user can call malloc() to allocate the buffer since continuous physical memory is not needed. Return values are as follows: RETCODE_INVALID_PARAM means that the given argument parameter, param is invalid. It has a null pointer, or addr in EncReportInfo is a null pointer when enable is 1.
DEC_SET_REPORT_ USERDATA	param is a pointer to an DecReportInfo. addr cannot be a null pointer and size cannot be zero when the enable flag is 1, so the user needs to allocate memory. The user can call malloc() to allocate the buffer since continuous physical memory is not needed. Return values are as follows: RETCODE_INVALID_PARAM means that the given argument parameter, param is invalid. It has a null pointer, or addr in EncReportInfo is a null pointer when enable is 1. DEC_SET_FRAME_DELAY
DEC_SET_FRAME_DELAY	HOST can set the number of frames to be delayed before display (H.264/AVC only) by using this command. This command is useful when display frame buffer delay is supposed to happen for buffering decoded picture reorder and HOST is sure of that. Unless this command is executed, VPU has display frame buffer delay as frameBufDelay value of DecInitialInfo structure.

4.9 i.MX 6 VPU Control

This section explains how VPU works, and provides few example applications that can be used with i.MX 6 VPU API.

This section describes the VPU control scheme based on the API functions and includes some practical programming issues.

4.9.1 VPU Initialization

When the host processor enables the VPU for the first time, the following initialization process should be performed. These operations are completed by calling a single API function, **vpu_Init**().

- Disable the BIT processor by setting BIT_CODE_RUN (BASE + 0x000) = 0

- Write the BIT processor microcode to the SDRAM accessible by the VPU during run-time
- Download the first *N* Kbytes of microcode to the BIT processor code memory
- Set the BIT processor buffer pointers, working buffer, parameter buffer and code buffer
- Set the stream buffer control options and the frame buffer endian mode
- Enable interrupt and reset registers
- Enable the BIT processor by setting BIT_CODE_RUN register = 1
- Wait until **vpu_IsBusy**() returns RETCODE_IDLE

Detailed information about each of these initialization steps and some programming tips are presented in the following sections.

4.9.1.1 Version Check of BIT Processor Microcode

The application can check the version information of the BIT processor microcode during runtime. The version number of microcode is a 32-bit value. The 16 most significant bits are the internal product number, and the 16 least significant bits are the version number specified by the following rule:

- Bits 15:12 = Major revision
- Bits 11:8 = Minor revision
- Bits 7:0 = Revision patches

This version number can have a value from 0.0.0 to 15.15.255. A dedicated command, **vpu_GetVersionInfo**(), is used for this version check and is supported after initialization.

4.9.1.2 BIT Processor Enable and Disable

The BIT processor has a dedicated register that activates or deactivates the BIT processor during run-time, BIT_CODE_RUN (BASE + 0x000). During initialization, the BIT processor program memory is updated and some configuration registers for controlling VPU operations are also set. During this process, the BIT processor should be disabled. After finishing the initialization process, the host processor enables the BIT processor. Then the BIT processor starts its own internal initialization process and is ready for operation.

4.9.1.3 BIT Processor Data Buffer Management

The BIT processor requires a certain amount of SDRAM space for its codec operations. This dedicated memory space includes memory space for the BIT processor microcode, internal work buffer, parameter buffers, and so on. The size of each sub-buffer as follows:

```
#define CODE_BUF_SIZE (132*1024)     // byte size of Code buffer
#define WORK_BUF_SIZE (256*1024)     // byte size of Work Buffer
#define PARA_BUF_SIZE (8*1024)       // byte size of Parameter Buffer
```

In VPU API, the initialization function only receives the start address of this internal buffer as an argument. Therefore, the total sum of the VPU processing buffer space, starting from the start address, should be dedicated memory space for VPU and no other process should access this memory space while VPU is enabled. It is highly recommended for the host processor to reserve the specified size of the dedicated buffer for the BIT processor and call **vpu_Init**() with the start address of the reserved memory. The start addresses of internal buffer partitions, code buffer, work buffer, and parameter buffer are calculated inside **vpu_Init**() function and the calculated start addresses are set in the host interface.

In addition to the above sub-buffers, VPU requires buffers for saving SPS/PPS and SLICE RBSP when decoding a H.264 stream. In general, 5 Kbytes is sufficient for the SPS/PPS save buffer and a quarter of the raw YUV image size is sufficient for the SLICE save buffer. If VPU requires more buffer space to decode a H.264 stream, VPU reports a buffer overflow.

4.9.1.4 BIT Processor Microcode Management

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

Since the dynamic reloading is completed by the VPU itself, the host processor only needs to copy the given microcode to the reserved code buffer before initializing the VPU. Of course, the first loading of the microcode to the BIT processor program memory should be completed separately by the host processor.

4.9.1.5 Stream Buffer Management

The stream buffer is a shared buffer between the host processor and the VPU for exchanging stream data. There are two different streaming schemes for decoding: ring-buffer and line-buffer. The ring-buffer scheme is used for host applications to reserve a fixed size of memory space and use it during codec operations. On the other hand, the line buffer scheme is used for host application to allocate a stream buffer dynamically and use it frame-by-frame.

The host processor also can choose the endian option of the stream buffer and can enable or disable the buffer full/empty check option. All these options for stream buffer data management are stored in a dedicated host interface register, BIT_BITSTREAM_CTRL, and are referenced by the BIT processor during run-time.

For decoding, the VPU provides both streaming options. But sometimes multiple-instance decoding may require a different streaming option for each decoder instance. For example, while playing a local video file, the application might need to decode a digital video broadcast. In this case, the different types of streaming mode can be helpful for the application design and the different streaming option is applied to each decoder instance independently.

4.9.1.5.1 Ring-Buffer Scheme (Packet Mode)

The ring-buffer scheme is preferred in packet-based video communication and streaming applications. In packet-based streaming based on a ring-buffer, the read and write pointers automatically wrap around at the boundaries. When the application downloads a new chunk of the bitstream, the application should check the available space in the bitstream buffer. Even though the available space can easily be calculated from the read pointer, write pointer and buffer size, the VPU API provides a dedicated function for providing the buffer read pointer, buffer write pointer and the available space in the stream buffer, **vpu_DecGetBitStreamBuffer**(). Based on the returned value from this API function, the application downloads a new chunk of bitstream data whose size should be smaller than the available buffer space. The amount of bits transferred into the stream buffer should be notified to the VPU using **vpu_DecUpdateBitStreamBuffer**().

4.9.2 Interrupt Signaling Management

To achieve maximum efficiency in VPU control, the VPU IP provides interrupt signaling for completion of a requested operation as well as stream buffer empty/full. For some commands with a quick return, interrupt signaling is not helpful so interrupt signaling is not provided.

The VPU provides interrupt signaling for the following commands:

- BIT_RUN_COMPLETE-BIT processor initialization complete after setting BIT_CODE_RUN
- DEC_SEQ_INIT-Decoder sequence initialization complete
- DEC_SEQ_END-Decoder sequence termination complete

- DEC_PIC_RUN-Decoder picture processing complete
- DEC_SET_FRAME_BUF-Decoder frame buffer registration complete
- DEC_PARA_SET-External header syntax transfer to decoder complete
- DEC_BUF_FLUSH-Flushing decoder stream buffer complete

DEC_SEQ_INIT and DEC_PIC_RUN can cause VPU to stall when the input bitstream is not large enough. Therefore, enabling the bitstream buffer-empty interrupt with these two interrupts, avoids unnecessary cycle consumptions in the host application. Each interrupt is easily enabled or disabled by writing 0 or 1 to the corresponding bit field of interrupt enable register. When an interrupt is signaled, the application checks the source of the interrupt by checking the value of interrupt reason register. When interrupt signaling is not easily applicable, these interrupts can be replaced by a polling scheme by reading the BIT processor busy-flag.

Note: *Only the DEC_PIC_RUN interrupt is used by applications. The other interrupts are used internally by the API or not supported.*

4.10 Encoder Control

4.10.1 Creating an Encoder Instance

After initializing VPU, an application creates an encode instance and acquires a handle for specifying that encoder instance is the first step to run an encoder operation. This is accomplished using a single API function called **vpu_EncOpen**().

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

- Bitstream buffer address and size-Physical address of the bitstream buffer start and its size.
- Codec standard-Video codec standard such as H.263, MPEG-4, H.264 or MJPEG.
- Picture size-Picture width and height.
- Target frame rate and bitrate with Video Buffer Verifier (VBV) model parameters, initialDelay and vbvBufferSize-VBV mode parameters are optional even when rate control is enabled.
- Gop size-Frequency of periodic intra (or IDR) pictures in the encoded stream output.
- Slice enable/disable, slice size mode and slice size-Slice mode enable or disable as well as the slice size and size mode (number of bits or number of Mbytes in each slice).
- Output report such as sliceReport, mbReport, and qpReport, and so on. qpReport option is only supported in H.263/MPEG-4 encoders meaning informative output data such as slice boundary, MB boundary in encoded bitstream.
- Miscellaneous options such as enableAutoSkip and intraRefresh-Enable auto-skipping of pictures when the output bit count is large enough as well as enable intra-refresh for error robustness and the number of intra MB in a non-intra picture.
- Ring buffer mode enable, allows streaming mode setting for each encoder instance independently-Application decides whether a ring-buffer based streaming scheme is used or not. When this option is disabled, a frame-based streaming scheme is used with a line-buffer scheme.
- Intra quantization step-Intra Qstep value is configurable by specifying this value greater than 0. Even if rate control is enabled, the VPU encoder uses this fixed quantization step for all I-frames. This intra quantization step is 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, and FMO parameters in H.264, source chroma format and thumbnail parameters and table coefficients in MJPEG and so on.

Using these options, the application receives a well optimized output for the requirements of the target application. Some output information options such as sliceReport, mbReport, qpReport, and so on, help application developers satisfy the constraints for target applications.

For example, for a fixed packet size, an application might need to insert one slice to a certain amount of bits. If the 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 the encoding process. Therefore, the application divides the slice into two packets which causes an inefficiency in the packetization. To achieve an easy packetization, the application sets the slice size to (packet_size - *N*) with a certain margin of *N*, which allows the output slice size to be less than the packet size. Then the application easily adds a slice into a packet by referring to the slice boundary information provided by the VPU as encoder output.

MJPEG can be encoded with various YUV format such as 4:4:4 by setting source format variable. 4:0:0, 4:2:0, 4:2:2 horizontal/vertical and 4:4:4 formats are supported in i.MX 6 MJPEG encoder. i.MX 6 VPU also supports encoding by using a user-defined Huffman Table and Q matrix. To encode by using a user-defined Huffman Table and Q matrix, the host must save the coefficients in a pre-defined format and set the pointer to the area.

After creating an encoder instance with these parameters, the application cannot change these parameters. If the application wants to change any of these basic parameters, it should close this instance and re-create another encoder instance with new initial parameters. However, the application may need to change some of these initial parameters depending on the target application environment. Using the dynamic configuration command, the VPU API enables the application to configure part of these initial parameters dynamically. For details, refer to Section 4.7.9.

The API function, **vpu_EncOpen**(), does not require any operations on the VPU side. Instead, it declares all of the internal parameters used in later stages as well as the bitstream buffer information.

4.10.2 Configuring VPU for Encoder Instance

4.10.2.1 Sequence Initialization

After registering all of the required information for the new encoder instance, the host application configures VPU to support the new encoder instance. This procedure is completed by setting the encoder related information in the VPU host interface registers and giving a command, ENC_SEQ_INIT, to VPU for initiating the internal configuration operation.

This process is mainly completed by an API function, **vpu_EncGetInitialInfo**(). This function returns a crucial output parameter for encoder operations and the minimum number of frame buffers. Normally, this process does not require much time, and it should be done only once at the beginning of each encoder instance. Therefore, it is not recommended to use an interrupt signal for this function. Interrupt signaling is allowed, however, after completion of this operation by enabling the corresponding bit on interrupt enable register.

4.10.2.2 Registering Frame Buffers During Configuration Process

The configuration process is completed by registering the frame buffers in VPU for picture encoding operations. In this final stage of configuration, the parameter, minimum number of frame buffers, returned from **vpu_EncGetInitialInfo**(), has an important meaning. This parameter means that the application should reserve at least the same number of frame buffers for VPU for proper encoding operation. For MJPEG, the frame buffer is not necessary, because MJPEG does not need motion compensation. Therefore, only the frame buffer stride is transferred to VPU in this stage. The stride value is used as the stride of the source image frame buffer.

4.10.2.3 Generating High-Level Header Syntaxes

Automatic header syntax generation (such as VOL in MPEG-4, SPS/PPS in AVC) is not supported.

When the encoder instance has been opened by calling **vpu_EncGetInitialInfo**(), the application generates the high-level header syntaxes such as VOS/VO/VOL headers in MPEG-4 and SPS/PPS in AVC from the VPU

using **vpu_EncGiveCommand**(). These high-level syntaxes can also be used directly for negotiation in the transport protocol layer of the application.

There are two possible methods for generating these header syntaxes: by PARA_BUF or by the stream buffer. The recommended way for generating the header syntaxes is to use the ENC_PUT_AVC/MP4 _HEADER command by the stream buffer. If the application uses this set of commands, the resulting header syntaxes are stored into the bitstream buffer according to the given endian setting.

If DecBufReset is enabled, the output header syntaxes are written to the bitstream buffer starting from the base address of the bitstream buffer. If the application does not read out each header syntax one-by-one, they are overwritten by the following header syntaxes. If the application wants to read out a set of header syntaxes (such as VOS/VO/VOL or SPS/PPS), then the application should disable DecBufReset and enable the DecBufFlush bit. After completing the generation of the last header syntax, the application can read out a cascaded set of header syntaxes together.

The other method for generating header syntaxes, by PARA_BUF, is used when the application wants to generate header syntaxes in the middle of encoding. It can be accomplished using ENC_GET_XXX_HEADER for MPEG-4, and ENC_GET_XXX_RBSP for AVC. Regardless of the streaming mode, this command generates header syntaxes successfully, but the endian setting is always big endian. So for little endian systems, an endian conversion should be performed.

4.10.3 Running Picture Encoder on VPU

4.10.3.1 YUV Input Loading

Before running a picture encoder operation, the host application should provide a 4:2:0 or 4:2:2 vertical formatted input YUV image with a pre-defined size for H.263, MPEG-4 and H.264. The host should provide 4:2:0, 4:2:2 vertical/horizontal, 4:4:4 or 4:0:0 formatted input YUV for MJPEG. If the input image is coming from an external video input device, such as a CMOS sensor, the VPU idles while waiting for completion of the receiving input picture. To avoid this idling, use a dual buffering scheme for the input image so that the encoder does not spend any cycles idling before starting operation.

4.10.3.2 Initiating Picture Encoding

When activating picture encoding operations, the application provides the following information to the VPU:

- Source frame address-Base address of each component of input YUV picture
- Quantization step-for the current picture which is ignored when rate control is enabled
- Forced frame skip and forced I-picture options-Forced frame skip is skipping the current frame encoding unconditionally and force I-picture is encoding current frame as I-frame unconditionally
- Source format-The VPU supports 4:2:2 vertical format source image. The source image is converted to 4:2:0 format automatically

After providing this information to the VPU, the host processor initiates a picture encoding operation by sending a ENC_PIC_RUN command to the VPU.

These processes can be performed by calling a single API function, **vpu_EncStartOneFrame**() with the EncParam structure. This API function initiates a picture encoding operation. Return from this API does not mean that picture encoding is completed, only that the encoding operation began successfully.

The quantization step size given to the VPU with ENC_PIC_RUN is only meaningful when the rate control option is disabled. This additional feature is provided to support application-specific VBR encoder operations.

The forced frame skip option is used when encoding a new picture is not allowed temporarily. Automatic frame skipping in the VPU rate control is used for limiting the output amount of the bitstream under the given target bit-rate. Also, the forced frame skip can be used by the application when encoding a picture is problematic under certain external situations, for example, if the channel condition is temporarily unacceptable and transmitting the

encoded stream is impossible. Then the application can suspend the encoder operation for a while using this forced frame skip option.

The forced I-frame option is used 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 the decoder side, the best way to recover from a decoder error is to send an I-frame. Using this forced I-frame option, the application can achieve error-recovery of the remote receiver side very effectively.

4.10.3.3 Completion of Picture Encoding

The application can be completing other tasks while waiting for the completion of picture encoding operation, such as packetization of the encoded stream for transmission. The application can use two different type of schemes for detecting completion of the picture encoding operation: polling a status register or interrupt signaling. When the application is using a polling scheme, the application checks the BusyFlag register of the BIT processor. Calling **vpu_IsBusy**() gives the same result.

Interrupt signaling can be the most efficient way to check the completion of a given command. An interrupt signal for the ENC_PIC_RUN command is mapped on bit 3 of the interrupt enable register. Therefore, the application can use this dedicated interrupt signal from VPU to determine the completion of the picture encoder operation.

4.10.3.4 Encoder Stream Handling

When the encoder stream buffer is large enough to store any size of picture stream, the encoder does not need to retrieve any bitstream data during the picture encoder operation. After the encoder operation is complete, the host application reads the encoded bitstream according to the requirements of packetization.

When the encoder stream buffer is not large enough to store a complete picture stream, the encoder buffer-full occurs and until this buffer-full situation is resolved, the encoder task running on the VPU is stalled. Therefore, while the picture is encoding, the application should continue reading out the encoded bitstream from stream buffer to avoid this stalling.

When using a ring-buffer scheme with a limited size of encoder stream buffer, stream reading during encoder operation is recommended. Using two dedicated functions, **vpu_EncGetBitStreamBuffer**() and **vpu_EncUpdateBitStreamBuffer**(), the application can easily handle the read pointer while accessing the encoder bitstream buffer. If the ring-buffer option is disabled with a stream buffer large enough to store one encoded picture data, the host can wait to read the encoded bitstream at the end of each picture encoding. In this case, the application can safely complete other tasks while the picture encoding is running on the VPU. The **vpu_EncGetBitStreamBuffer**() and **vpu_EncUpdateBitStreamBuffer**() functions have no meaning when the application uses the frame-based streaming option.

4.10.3.5 Acquiring Encoder Results

When picture encoding is complete, the host application retrieves the encoded output such as the encoded picture type, number of slices, and so on. According to the input parameter settings of the picture encoding, the slice boundary and MB boundary information can also be acquired from VPU. For H.263/MPEG-4 decoding, the MB Qstep information can be acquired from VPU. This encoder output information is generally placed on the parameter buffer with predefined formats (for the predefined formats of the output information, see the following documents:

- *i.MX 6Dual/6Quad Applications Processor Reference Manual* (IMX6DQRM)
- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)

Therefore, the application can read out this information directly from the parameter buffer using the base address of each data structure.

VPU API provides a function for retrieving the output results of the picture encoder, **VPU_EncGetOutputInfo**(), which has a output data structure that includes the following information:

- Start address of encoded picture and its size
- Number of slices in the encoded picture
- Slice boundary information in the encoded bitstream
- MB boundary information in the encoded bitstream
- Application-specific information for packetization such as MB Qstep information

Some packetization schemes, such as Real-time Transfer Protocol (RTP), require some internal information of encoded picture depending on the codec standard.

The slice information is useful for packet-based applications which have limitations of the slice start in the video packet. The slice information is also useful for implementing slice re-ordering on the application side such as Arbitrary Slice Ordering (ASO) in the H.264 standard.

VPU API includes a constraint on using the encoder initiation function and the encoder result acquisition. When using VPU API, the 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 is not initiated by calling **vpu_EncStartOneFrame**(). Most VPU commands are not allowed unless the application calls **VPU_EncGetOutputInfo**() after completion of the picture encoding operation. This constraint is used to protect the encoded results from being overwritten from another thread by mistake in a multi-instance environment. Therefore, the application should regard the **vpu_EncGetOutputInfo**() function as a releasing command of the VPU from the current picture encoding operation.

4.10.4 Terminating an Encoder Instance

When the application finishes with the encoder operation and terminates an encoder instance, the application releases the handle of this instance to inform the VPU that this instance is terminated by giving the SEQ_END command to the VPU. This can be accomplished by calling **vpu_EncClose**() function.

4.10.5 Dynamic Configuration Commands (picture encoding operations)

While running sequential picture encoding operations, the application may need to give special commands to VPU such as rotating the input pictures before encoding, inserting a high layer header syntaxes, and so on. The VPU API provides a set of commands to support the following special requests from the host application:

- Rotate and mirror source frame before encoding.
- Extract high layer header syntaxes such as VOS/VO/VOL in MPEG-4, and SPS/PPS in H.264 for external use.
- Insert high layer header syntaxes such as VOS/VO/VOL in MPEG-4 and SPS/PPS in H.264.
- Change encoder parameters such as bitrate, frame rate, GOP number, and slice mode dynamically between picture encoding operations.

4.11 Decoder Control

4.11.1 Creating a Decoder Instance

After initialization of VPU, the next step to run a decoder operation is to create a decoder instance and acquire a handle for specifying that decoder instance. This is accomplished using a single API function, **vpu_DecOpen**().

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

- Bitstream buffer address and size is a physical address of bitstream buffer start address and its size.
- Codec standard is a video codec standard such as H.263, MPEG-4, H.264 or VC-1.
- MPEG-4 deblocking filter enable is enable or disable MPEG-4 de-blocking filter option.
- ReorderEnable-Enable or disable H.264 display reordering option. This option is ignored for other decoder standards. It should usually be set to 1.
- SPS/PPS RBSP save buffer address and size is a physical address and size of buffer for SPS and PPS.
- Enable thumbnail decoding of MJPEG-Enable thumbnail decoding. If the host enables thumbnail decoding, the decoded output is s thumbnail.

For decoding, most information is acquired from the input stream, so there are few required parameters for creating a decoder instance. 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 the bitstream buffer information.

4.11.1.1 AVC Display Reordering

The AVC-specific display reordering option should be used carefully, because it drastically varies the behavior of the AVC decoder. In principle, this option should always be enabled because the flag for this option is embedded in the header syntax. According to the options in the header, the required frame buffer size is automatically determined by the VPU.

When creating a decoder instance for H.264, the application should decide if display reordering is used. In principle, this bit field should be set to 1, because the display reordering option is enabled or disabled automatically according to the values of the corresponding header fields. But in practice, there are too many streams which do not actually use display reordering but display reordering option is enabled.

Display reordering generally requires many more decoder buffers, a much longer delay, and some complex constraints in decoder operations. When display reordering is not used even though the display reordering option is enabled on the baseline profile stream, the application can force the VPU decoder to ignore this option and a flag is provided for this case.

When this option is disabled, the minimum number of frame buffers is reference frame number + 2. Whenever one frame decoding is complete, a display (or decoded) output is provided from the VPU, so the decoder operation is the same as a normal decoder operation.

But when this option is enabled, the minimum number of frame buffers is MAX(reference frame number, 16) + 2 for the worst case. After decoding one frame, the VPU cannot provide a display output because display order can be different from the decoding order. In the worst case, the first display output is provided from the VPU after decoding 17 frames. Because of this characteristic of display reordering, the VPU AVC decoder always decodes display delay + 1 frames during the first call of the picture decoding when display reordering is enabled in the stream.

In practice, there are many streams which do not use display reordering, but the flag in the header is enabled. In this case, the host application must allocate unnecessarily more frame buffers and apply large delays. Considering this practical cases, this option for forced-disable of display reordering is provided in the VPU API.

4.11.2 Configuring VPU for Decoder Instance

4.11.2.1 Feeding Bitstream into Stream Buffer

For the 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. To start sequence initialization, the application fills the decoder stream buffers with enough bitstream data. In some applications, the host

applications cannot guarantee that those kinds of header syntaxes are placed at the beginning of the bitstream. In this case, until the VPU successfully receives all of the required information from the input stream, the application should keep feeding the input data stream to the decoder bitstream buffer.

To feed the input bitstream, the host application should know the available space in the bitstream buffer. This is determined using the read pointer, write pointer and stream buffer size because the stream buffer operates as a ring-buffer. Getting the available space in the stream buffer, the application can directly download the decoder input stream to the bitstream buffer. After completing the stream download, the application informs the amount of downloaded stream data by updating the stream write pointer.

The VPU API provides an API function to get the stream read pointer, write pointer and available space, **vpu_DecGetBitstreamBuffer**(). Updating the write pointer is accomplished using the API function, **vpu_DecUpdateBitstreamBuffer**().

4.11.2.2 Sequence Initialization when configuring VPU for Decoder Instance

After creating a new instance and feeding the input bitstream to the stream buffer, the application gives the DEC_SEQ_INIT command to the VPU to get the decoder configuration information from the bitstream. After parsing the header syntaxes, the decoder returns the following crucial information about the decoder configuration:

- Picture size-Picture width and height
- Frame rate-Decoder frame rate
- Picture cropping rectangle information-Information about H.264 decoder picture cropping rectangle which is the offset of top-left point and bottom-right point from the origin of frame buffer
- Minimum number of frame buffers
- MPEG-4 option information-Enable or disable 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 the deblocking option of the H.263 decoder is enabled or disabled. When the external post-deblocking filter is used for H.263, this flag is used to avoid repetition of the 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 are used in next decoding after decoding one frame
- Estimated slice save buffer sizes-The size of the slice save buffer. The VPU reports two different sizes: recommended and worst-case
- MJPEG thumbnail enable information-This flag indicates whether thumbnail image of MJPEG exists or not. When thumbnail does not exist in the stream, the VPU returns failure if the host application enables the thumbnail decoding option
- MJPEG image YUV format-Image YUV format. The host must allocate frame buffer by this value

The picture size acquired from the bitstream might not be a multiple of 16x16. However, to perform the decoder operation properly, frame buffer size should be a multiple of 16x16. Therefore, the returned size is modified to be a multiple of 16x16 after a ceiling operation. Using the picture size and the minimum number of frame buffers, the application reserves frame buffers and provides them to the VPU before starting the picture decoding operation.

The frame buffer delay is an H.264-specific parameter for supporting display reordering. If the application supports display reordering and reordering requires five additional frame buffers, for example, then the first display output comes out from decoder after decoding the 6^{th} frame. Theoretically, the maximum delay for display reordering is a 16-frames.

The VPU API provides a function to handle the DEC_SEQ_INIT operations, **vpu_DecGetInitialInfo**(). Completion of this function is signaled by a dedicated interrupt or by polling the BusyFlag.

An important issue in SEQ_INIT operation is error-handling because any errors in the high layer header syntaxes cause serious problems in decoding operations. Generally, many marker bits are added to the header syntaxes to assist error detection. When header syntaxes included in the stream have crucial errors, or when header syntaxes are not received for a long time, the VPU can be stuck on this task and no other instances can run on the VPU. Therefore, the VPU API provides a special function which is used in this situation, called **vpu_SetSeqInitEsc**(). When this function is called and the stream buffer is empty, the VPU automatically terminates the SEQ_INIT operation. Then the host application decides whether to close this instance or retry 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 the **vpu_SetSeqInitEsc**() function again. This flag affects all the decoder instances performing a DEC_SEQ_INIT operation.

4.11.2.3 Registering Frame Buffers

This configuring process is completed by registering the frame buffers to the VPU for picture decoding operations. In this final stage of configuration, the parameter returned from **vpu_DecGetInitialInfo**(), the minimum number of frame buffer, has an important meaning. This parameter means that the application should reserve at least the same number of frame buffers to the VPU for proper decoding operation.

The size of the frame buffers is calculated from the picture width and height. When both the picture width and height are a multiple of 16, the picture size is the size as the frame buffers. If both the picture width and height are not a multiple of 16, the application should apply a ceiling operation to the picture width or picture height to get the smallest multiple of 16 larger than picture width or picture height.

In addition to registering the frame buffers to the VPU, the slice save buffer is also registered in this step. The recommended buffer size is given by calling **vpu_DecGetInitialInfo**().

4.11.3 Running Picture Decoder On VPU

4.11.3.1 Initiating Picture Decoding

When activating a picture decoding operation, the application provides the following information to the VPU:

- I-Frame Search Enable is enable or disable I-(IDR for H.264) frame search option.
- Frame Skip Mode is enable or disable skipping bitstream for the next frame decoding.
- DispOrderBuf-Enable or disable the next display output without decoding.

After providing these parameters to VPU, the application starts the picture decoding operation by sending a DEC_PIC_RUN command.

The pre-scan option is a special option for scanning the bitstream buffer to check if a full picture stream exists in the stream buffer. This option allows the application to determine whether the bitstream empty and decoder stalls or not before running the actual decoder operation. When this option is enabled and there is not a full picture stream in the decoder buffer, the DEC_PIC_RUN command does not initiate the picture decoding operation and returns immediately. Then, the application decides whether to retry the picture decoding after feeding more bitstream data or to handle other tasks for a while.

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

When display reordering in H.264 is enabled, the first decoded output is only available after decoding many frames. To avoid this, a constraint is added to the H.264 decoder that requires the decoder to fill all the reordering display buffers at the first time of picture decoding. That means, if the frame buffer delay received

from the stream header is five, the H.264 decoder should decode six frames at once at the first DEC_PIC_RUN operation. Then, the picture decoding always provides a picture output to be displayed. In this scenario, the pre-scan might cause problems, because it is designed for the case of one picture decoding. When display reordering is enabled, it is recommend that the first DEC_PIC_RUN be performed with pre-scan disabled.

To support display reordering in H.264 mode, a special parameter is used to flush the stored decoder output from the display reorder buffer without picture decoding. This option is designed for flushing out the decoded picture not yet displayed at the end of the decoding video sequence. When the display reordering option is enabled and the reordering frame buffer stores five decoded pictures, the first display output is available after the 6^{th} frame decoding. Therefore, at the end of the stream decoding, there are five decoded pictures which are not displayed yet even though there is no more available bitstream data to decode. In this case, the application may ignore these five non-displayed pictures or display them by setting the dispReorderBuf parameter to 1 and sending the DEC_PIC_RUN command until the VPU returns the decoded picture index of -1.

VPU API provides an API for handling all these complex operations, **vpu_DecStartOneFrame**(), which initiates the picture decoding operation and returns as soon as picture decoding has started on the VPU. Completion of picture decoding is checked using a different method.

4.11.3.2 Frame Skipping Option

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

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

This frame skip feature can be used by the application when the system performance is temporarily degraded and video decoding is significantly delayed. In this case, it is recommended for the application to use the I-(IDR in H.264 case) frame detect option. Using this option, the application can only decode I-(or IDR) frame properly without displaying erroneous 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 problems with AV synchronization.

In the random access case, the I-frame search option can be useful when the keyframe information in the file container is incorrect.

4.11.3.3 I-Frame Search for Random Access and Trick Mode

When a media player application is designed, trick modes and random access may be desirable features. To achieve these operations the application, decoder should support a feature for searching the I-frame in the middle of the decoder bitstream.

The I-frame search option is accomplished by setting the iframeSearchEnable of DecParam. The number of I-frames skipped is also set by setting skipframeNum of DecParam. (The same skipframeNum of DecParam is used for specifying the skipped frame number in frame skipping and I-search; however, the meaning of this value is somewhat different.) If skipframeNum = N, all the intermediate frames before the $(N+1)^{th}$ next I-frame

are skipped. This multiple I-frame skipping might be used for high-speed playback such as fast forward. By increasing the number N, the application can increase the speed of the fast forward. This kind of fast forward operation depends on the frequency of the I-(IDR) frames in the decoder input bitstream. Therefore, this type of trick mode can be applicable to applications specifying the maximum interval between I-frames.

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

1. Freeze the display and reset the decoder bit-stream buffer
2. Read the bitstream from the new file read pointer and transfer it into the decoder
3. Enable I-Search and run the picture decoding operation
4. If the buffer empty interrupt is signaled, feed more bitstream and wait for decoding completion
5. If decoding completion is detected, read the decoder results and resume display

Resetting the bitstream buffer in Step 1 can be accomplished by calling **vpu_DecBitBufferFlush**(). Starting the decoder operation with I-frame search can also be accomplished by calling **vpu_DecStartOneFrame**() with iframeSearchEnable of DecParam set to 1. The number of skipped frames specified by skipframeNum of DecParam is given by 1 in random access operation. When an interrupt of decoder completion or non-busy state of the BIT processor is detected, the I-frame is searched and decoded.

When the application uses the I-frame search option, the decoder should skip many bits in the decoder stream buffer. Therefore, the pre-scan option can be meaningless when used simultaneously with the I-search. In the VPU firmware; therefore, the pre-scan option is automatically disabled and settings for the pre-scan option are ignored. The application should handle stream buffer filling until the end of the I-search operation. Larger stream units are recommended in this case; otherwise, too many stream buffer empty interrupts might occur from the VPU side.

4.11.3.4 Decoder Stream Handling

When the decoder stream buffer includes a full picture stream, the host application does not need to worry about streaming in the middle of the decoder operation. Using the pre-scan option, the application can determine the status of the bitstream buffer in advance. If there is no full picture in the stream buffer, the application might feed more stream data to the stream buffer and start the picture decoding operation.

The VPU API provides an API function to get the stream read pointer, write pointer and available space in one function call, **vpu_DecGetBitstreamBuffer**(). The application can get the information about the available space in the stream buffer using this API and transfer an amount of stream data to the stream buffer which is less than or equal to the available size. When transferring the stream data, the application should take care of the end of the stream buffer to avoid unexpected data corruption. When transferring stream data to the stream buffer and the write pointer reaches the end of the stream buffer, the application should wrap the write pointer around to the beginning of the stream buffer and then continue downloading to avoid data corruption.

Updating the write pointer is accomplished using, **vpu_DecUpdateBitstreamBuffer**(). The write pointer wrap-around and updating of the write pointer is done by this API function by providing the downloaded stream size. Before updating the write pointer, the host application must finish transferring the stream data to the stream buffer. If not, a mismatch in access time may cause problems in the decoder operation.

4.11.3.5 Completion of Picture Decoding

Picture decoder operations take a certain amount of time, and the application can complete other tasks while calling **vpu_WaitForInt**() to wait for the completion of the picture decoding operation, such as display processing of the previously decoded output. The application can use two different schemes for detecting the completion of the picture decoding operation: polling a status register or waiting for an interrupt signal. When

the application uses the polling scheme, the application checks the BusyFlag Register of the BIT processor. Calling **vpu_IsBusy**() gives the same result.

Interrupt signaling can be the most efficient way to check the completion of a given command. An interrupt signal for the DEC_PIC_RUN command is mapped to bit 3 of the interrupt enable register. So the application can easily determine the completion of the picture decoder operation from this dedicated interrupt signal from the VPU.

4.11.3.6 Acquiring Decoder Results

When picture decoding is complete, the host application retrieves the decoded output, such as the display frame index, decoded frame index, decoded frame picture type, number of error concealed MBs, Pre-scan result, and so on. The VPU API provides a function for retrieving the output results of the picture decoder, **vpu_DecGetOutputInfo**().

The VPU API includes a constraint on using the decoder initiation function and decoder result acquisition. When using the VPU API, the 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 is not initiated by calling **vpu_DecStartOneFrame**(). This constraint is used to protect the decoded results from being overwritten from other thread by mistake in multi-instance environment. Therefore, the application should regard **vpu_DecGetOutputInfo**() function as a releasing command of the VPU from the current picture decoding operation.

4.11.3.6.1 Reading Display Output

The display frame index, indexFrameDisplay, is used to represent the frame buffer number where the display output picture is stored. It always equals the frame buffer index to be displayed. It can be different from the decoded picture index when display ordering control is enabled, such as display reordering of H.264, B-frame in VC-1, and so on.

At the beginning of sequence decoding, even after decoding several frames, there is no display output from decoder because of the order of display. For H.264 reordering, worst case scenario, the first display output can come out after the 17^{th} frame decoding. Therefore, at times there is no proper display buffer index. In this case, VPU decoder returns a negative frame buffer index for indexFrameDisplay of -3 or -2 depending on the frame skip option. Only at the end of sequence decoding is this value equal to -1 and the application can terminate the current decoder instance without any loss in picture display.

The table below shows the display output status based on the indexFrameDisplay values.

Table 5. indexFrameDisplay Values

indexFrameDisplay Value	Display Output Status
Non-negative value	Output index value points to the frame buffer index of the display output.
-1	Signals the end of sequence decoding, there is no more display output when the stream end is signaled to VPU.
-2	There is temporarily no display output because of the frame-skip option.
-3	There is temporarily no display output even without any action by the host application. Usually, this value occurs when an IDR picture is received for H.264 display-reordering mode.

4.11.3.6.2 Reading Decoded Output

The decoded frame index, indexFrameDecoded, is an optional output to the host application. This index is used to represent the frame buffer number where the decoded picture is stored. Usually, the host application does not

need to worry about this index. The display index, indexFrameDisplay, is sufficient to handle the output of the VPU decoder.

When there are not enough frame buffers to be written with decoded image data, this value is equal to -1 (0xFFFF). In this situation, the application re-calls **vpu_DecStartOneFrame**() after clearing the display flag by calling **vpu_DecClrDispFlag**().

When display ordering control is enabled for H.264 display reordering, VC-1 B-frame, and so on, at the end of sequence decoding, the host application needs to flush out the decoded frames for display. During this flushing operation, no actual decoding operations are performed. Under this situation, this value is equal to -1 (0xFFFF) to represent that there is no decoded frame this time. This negative decoded index is also used when picture decoding is skipped because of skip option or picture header error.

4.11.3.6.3 Reading Pre-Scan Result

The pre-scan result flag represents whether a full picture stream is included in the bitstream buffer before picture decoding. When this flag is equal to 0, the decoding operation is not performed because there is no full picture stream in the stream buffer. If application enables pre-scan and sets pre-scan mode to 0 (decoding a picture when full picture stream exists), the application should check this output parameter first to determine whether a decoding operation is performed or not.

When pre-scan result is 0 and the stream buffer is full and the current stream buffer is too small to store a full picture stream. To avoid dead-lock, the host application should disable the pre-scan option and re-run the picture decoding operation.

4.11.3.6.4 Display Cropping in H.264

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

4.11.3.6.5 Next Decoded Frame Index

Next decoded frame index, indexNextFrameDecoded[3], is an optional output to the host application. This indexes are used to represent the frame buffer index which is used in the next **VPU_DecStartOneFrame**() call. The 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, VC-1 B-frame, at the end of sequence decoding, the host application needs to flush out the decoded frames for display. During this flushing operation, no actual decoding operations are performed. In this situation, this value might be ignored.

4.11.3.6.6 Reading Lack of Additional Work Buffer

The VPU reports the status of the PS (SPS/PPS) save buffer and slice save buffer after it decodes one frame. If the VPU reports lack of PS save buffer, the VPU cannot properly decode the remaining input stream; therefore, it is best to close current instance in this situation. If the VPU reports lack of slice save buffer, the VPU can choose to either close and reopen the current instance or continue picture decoding regardless of display corruption until the next I-frame.

4.11.3.7 Management of Displaying Buffers Decoded

The VPU has flags to indicate if the frame buffer is displayed or not internally. The flag is set after the VPU returns the display frame index automatically and the VPU never uses the buffer for which the display flag is set. Before starting the decoding process, the VPU checks if there is a frame buffer available and returns immediately if there is no frame buffer to be written with decoded image with a current decoded index of -1. The host application clears the flag after completion of displaying the frame buffers by calling **vpu_DecClrDispFlag**().

4.12 Escape from Decoder Hang

Even when pre-scan is used, it is still possible for an application to experience decoder hanging because of a stream error or lack of available stream at the end of sequence decoding. In the middle of picture decoding, decoder hanging is signaled to the application through the decoder buffer empty interrupt if this interrupt is enabled, and the application can avoid decoder hanging by putting more bitstream data to stream buffer.

In some extraordinary cases and at the end of sequence decoding, the application avoids decoder hanging by means of garbage insertion or sending an end-of-stream command to VPU decoder. This is accomplished by calling **vpu_DecUpdateStreamBuffer**() with size of 0. As soon as VPU detects this setting, VPU terminates the current picture decoding with error concealment if applicable.

4.13 Terminating a Decoder Instance

4.13.1 Stream End and Last Picture in Stream Buffer

After the host application meets the end of stream and sends all of the stream data in the stream buffer, the host application must determine when the last picture output is coming out. If there is no display delay, this task is simple. But if display delay exists (reordering of the decoded pictures for display), this task might be difficult for the host application.

After sending the last byte of the stream data to bitstream buffer, host application must call vpu_DecUpdateBitstreamBuffer() with "size" = 0 to signal the end of stream to VPU thus prevent VPU from being stalled due to stream buffer empty, then keep calling vpu_DecStartOneFrame(). After the last display output picture has come out, the display frame index (indexFrameDisplay) will be changed to -1. When host application receives this index, it can easily detect the end of the sequence processing.

When display delay exists (display reordering option in H.264, B-frames in other codecs), host application gets the buffered decoder output frame even after finishing actual decoding operation. In this case, host application still needs to call vpu_DecStartOneFrame() as usual. Until the delayed display output frames are completely flushed out, the VPU decoder will provide the display frame index of the newly displayed output to the host application. And if there is no more available output, the VPU decoder returns a display frame index (indexFrameDisplay) of -1.

4.13.2 Closing Current Instance

When the application finishes the last picture decoding operation and terminates a decoder instance, the application releases the handle of this instance and inform the VPU that this instance is terminated by giving the SEQ_END command to the VPU. This can be accomplished by calling the **vpu_DecClose**() function.

4.14 Dynamic Configuration Commands

While running sequential picture decoding operations, application may need to give a special command to VPU. VPU API provides a set of commands to support the following special requests from the host application:

- Rotate and mirror output frame before decoding

- Apply SPS and PPS from the external out-of-band protocol
- Specify the frame buffer address for the MPEG-4 de-blocking filtered output

4.15 Example Applications

This section discusses the example applications provided for i.MX 6 VPU API.

4.15.1 VPU Library

VPU library and header file source code is located under Yocto Project build tmp work tree in imx-lib/*/vpu. The detailed source code structure of the VPU library and kernel space is presented in the Video Processing Unit (VPU) Driver chapter of the *i.MX Linux® Reference Manual* (document IMXLXRM).

The user may optionally configure the following following environment variables:

- VPU_FW_PATH-Directory where the vpu_fw_imx6q.bin or vpu_fw_imx6d.bin file is located. If this variable is not exported by the user, the vpu_fw_ mx6.bin file must be located in the /lib/firmware/vpu directory.

4.15.2 VPU Example Application

VPU example application is located under Yocto Project build tmp work tree in imx-test/*/test/mxc_vpu_test. This application gives an example of how to use the VPU API to control the VPU hardware to implement a decoder or an encoder. The following test cases are included in this test application:

- Decode streams to save to a YUV file or to display on an LCD.
- Encode streams from a YUV file or from camera captured data.
- Loopback-encode camera captured YUV data then decode it to a YUV and display on an LCD simultaneously.
- Network-encode camera captured YUV data and send it to another side to decode by UDP.

Note: *Only packet-based streaming mode with ring-buffer is included in this example application.*

Refer to the readme file for details about the usage of the application example. Section 4.15.2.1, and Section 4.15.2.2, describe the example applications usage for decoding streams to display on an LCD and encoding streams from camera captured data. These two examples are described in detail to illustrate how proper frame buffer management between VPU and V4L interface improves performance and avoids memory copy, especially memory for decoded YUV or captured YUV data.

4.15.2.1 Decode Stream to Display on LCD

The application should complete the following steps to decode streams to display on an LCD:

1. Call **vpu_Init**() to initialize the VPU. If there are multi-instances supported in this application, this function only needs to be called once.
2. Open a decoder instance using **vpu_DecOpen**(). Call **IOGetPhyMem**() before opening the instance to input oparam.bitstreamBuffer. Call **IOGetVirtMem**() to get the corresponding virtual address of the bitstream buffer, then fill the bitstream at this address in user space. Call **IOGetPhyMem**() for both the physical PS save buffer and physical slice save memory for H.264.
3. Call **vpu_DecGetBitstreamBuffer**() to get the bitstream buffer address to provide the proper amount of bitstream.
4. After transferring the decoder input stream, declare the amount of bits transferred into the bitstream buffer using **vpu_DecUpdateBitstreamBuffer**().
5. Get crucial parameters for decoder operations such as picture size, frame rate, required frame buffer size, and so on using **vpu_DecGetInitialInfo**(). Set escape to 1 by calling **vpu_DecSetEscSeqInit**(handle,

1) before this function is called. Set escape to 0 by calling **vpu_DecSetEscSeqInit**(handle, 0) after **vpu_DecGetInitialInfo**() is called.

6. Using the frame buffer requirement returned from **vpu_DecGetInitialInfo**(), allocate the proper size of the frame buffers and notify the VPU using **vpu_DecRegisterFrameBuffer**(). The requested frame buffer in PATH_V4L2 case to display the stream on the LCD is as follows:
   - Add two more buffers than minFrameBufferCount to the frame buffer count: **vpu_DecClrDispFlag**() is used to control if the frame buffer can be used for decoder again. One framebuffer dequeue from IPU is delayed for performance improvement and one framebuffer is delayed for display flag clear. Performance is better when more buffers are used if IPU performance is bottleneck.
   - Call **v4l_display_open**() to open the v4l device and request v4l buffers for image display. If VPU rotation or dering is enabled, larger frame buffers are needed. Two extra buffers are added in this example application. Register the first minFrameBufferCount + 2 buffers as bufY, bufCb, bufCr for the VPU decoder, and memory transfer is not needed for performance improvement. Call **IOGetPhyMem**() for bufMvCol part for VPU decoder usage.
   - Inform the VPU to register minFrameBufferCount + 2 buffers by calling **vpu_DecRegisterFrameBuffer**().
7. Start picture decoder operation picture-by-picture using **vpu_DecStartOneFrame**().
   - If rotation is enabled, the SET_ROTATION_ANGLE, SET_ROTATOR_STRIDE and ENABLE_ROTATION commands need to be given before starting decoding by calling **vpu_DecGiveCommand**(). The rotator stride is the picture height if the rotation angle is 90° or 270°; otherwise, the stride is the picture width.
   - If dering is enabled, the ENABLE_DERING command needs to be given before starting decoding.
   - If mirror is enabled, the SET_MIRROR_DIRECTION and ENABLE_MIRRORING commands need to be given.
   - Since there are two extra buffers used for rotation or dering, the SET_ROTATOR_OUTPUT commands need to be set before each picture decoder.
   - Start the picture decoder operation by calling **vpu_DecStartOneFrame**().
8. Wait for the completion of the picture decoder operation interrupt event by calling **vpu_WaitforInt**(). **vpu_IsBusy**() is used to check if the VPU is busy. If the VPU is not busy, go to the next step. Otherwise, wait again and more bitstream can be filled to the bitstreamBuffer while waiting.
9. Check the results of the decoder operation using **vpu_DecGetOutputInfo**(). Go to different case as defined by outputinfo. For example, -1 in outinfo.indexFrameDisplay indicates that the decoder completed. Values of -2 or -3 in outinfo.indexFrameDisplay indicates that no picture needs to be displayed. A positive value in outinfo.indexFrameDisplay indicates the displayed buffer index, and **v4l_put_data**() can be called to display the image on the LCD.
   In the **v4l_put_data**() function, IOCTL VIDIOC_QBUF is set to queue the buffer to the v4l module for display. Also, IOCTL VIDIOC_DQBUF is used to get one buffer that image has been displayed and can be used again for the decoder. Here, one frame buffer dequeue from the IPU is delayed, then the VPU and IPU operate in an asynchronous method for performance improvement.
10. After displaying the n^{th} frame buffer, clear the buffer display flag using **vpu_DecClrDispFlag**(). This function does not need to be called for the STD_MJPG codec. One frame buffer is delayed for display flag clear, that means, previous dequeued framebuffer index was cleared by the VIDIOC_DQBUF IOCTL.
11. If there is more bitstream to decode, go to step 7, otherwise go to the next step
12. Terminate the sequence operation by closing the instance using **vpu_DecClose**(). Make sure **vpu_DecGetOutputInfo**() is called for each corresponding **vpu_DecStartOneFrame**() call before closing the instance although the last output information may be not useful.
13. Free all memory that was allocate by calling **IOFreePhyMem**() and **IOFreeVirtMem**(). **v4l_display_close**() needs to be called to free all v4l related resource, including v4l buffers.
14. Call **vpu_UnInit**() to release the system resources before exit. If there are multi-instances supported in this application, this function only needs to be called once.

4.15.2.2 Encode Stream from Camera Captured Data

The application should complete the following steps to encode streams from camera captured data:

1. Call **vpu_Init**() to initialize the VPU. If there are multi-instances supported in this application, this function only needs to be called once.
2. Open a encoder instance using **vpu_EncOpen**(). Call **IOGetPhyMem**() to input encop.bitstreamBuffer for the physical continuous bitstream buffer before opening the instance. Call **IOGetVirtMem**() to get the corresponding virtual address of the bitstream buffer, then fill the bitstream to this address in user space. If rotation is enabled and the rotation angle is 90° or 270°, the picture width and height must be swapped.
3. If rotation is enabled, give commands ENABLE_ROTATION and SET_ROTATION_ANGLE. If mirror is enabled, give commands ENABLE_MIRRORING and SET_MIRROR_DIRECTION.
4. Get crucial parameters for encoder operations such as required frame buffer size, and so on using **vpu_EncGetInitialInfo**().
5. Using the frame buffer requirement returned from **vpu_DecGetInitialInfo**(), allocate the proper size of the frame buffers and notify the VPU using **vpu_EncRegisterFrameBuffer**(). The requested frame buffer for the source frame in PATH_V4L2 to encode camera captured data is as follows:
   - Allocate the minFrameBufferCount frame buffers by calling **IOGetPhyMem**() and register them to the VPU for encoder using **vpu_EncRegisterFrameBuffer**().
   - Another frame buffer is needed for the source frame buffer. Call **v4l_capture_setup**() to open the v4l device for camera and request v4l buffers. In this example, three v4l buffers are allocated. Call **v4l_start_capturing**() to start camera capture. Pass the dequeued v4l buffer address by calling **v4l_get_capture_data**() as encoder source frame in each picture encoder, then no need to memory transfer for performance improvement.
6. Generate the high-level header syntaxes using **vpu_EncGiveCommand**().
7. Start picture encoder operation picture-by-picture using **vpu_EncStartOneFrame**(). Pass dequeued v4l buffer address by calling **v4l_get_capture_data**() as the encoder source frame before each picture encoder is started.
8. Wait for the completion of picture decoder operation interrupt event calling **vpu_WaitforInt**(). Use **vpu_IsBusy**() to check if the VPU is busy. If the VPU is not busy, go to the next step; otherwise, wait again.
9. After encoding a frame is complete, check the results of encoder operation using **vpu_EncGetOutputInfo**(). After the output information is received, call **v4l_put_capture_data**() to the VIDIOC_QBUF v4l buffer for the next capture usage.
10. If there are more frames to encode, go to Step 7; otherwise, go to the next step.
11. Terminate the sequence operation by closing the instance using **vpu_DecClose**(). Make sure **vpu_DecGetOutputInfo**() is called for each corresponding **vpu_DecStartOneFrame**() call before closing the instance although the last output information may be not useful.
12. Free all allocated memory and v4l resource using **IOFreePhyMem**() and **IOFreeVirtMem**(). Call **v4l_stop_capturing**() to stop capture.
13. Call **vpu_UnInit**() to release the system resources. If there are multi-instances supported in this application, this function only needs to be called once.

4.15.3 Other Issues

Some important issues are as follows:

- Performance is better both on the VPU and IPU when chromainterleave mode is enabled.
- To avoid the VPU hanging if there is not enough stream data, enable prescan in networking mode to first scan the stream buffer. This flag can be disabled if the bitstream buffer is large in real video playback and the application can guarantee the bitstream buffer is sufficient.
- Since IPU rotation performance is better than the VPU, use IPU rotation and not VPU rotation.

5 Note About the Source Code in the Document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2023 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

6 Revision History

This table provides the revision history.

Revision history

Revision number	Date	Substantive changes
LF6.1.55_2.2.0	12/2023	Upgraded to the 6.1.55 kernel.
LF6.1.36_2.1.0	09/2023	Upgraded to the 6.1.36 kernel and added the i.MX 91P.
LF6.1.22_2.0.0	06/2023	Upgraded to the 6.1.22 kernel.
LF6.1.1_1.0.0	03/2023	Upgraded to the 6.1.1 kernel.
LF5.15.71_2.2.0	12/2022	Upgraded to the 5.15.71 kernel.
LF5.15.52_2.1.0	09/2022	Upgraded to the 5.15.52 kernel, and added the i.MX 93.
LF5.15.32_2.0.0	06/2022	Upgraded to the 5.15.32 kernel, U-Boot 2022.04, and Kirkstone Yocto.
LF5.15.5_1.0.0	03/2022	Upgraded to the 5.15.5 kernel, Honister Yocto, and Qt6.
LF5.10.72_2.2.0	12/2021	Upgraded the kernel to 5.10.72 and updated the BSP.
LF5.10.52_2.1.0	09/2021	Updated for i.MX 8ULP Alpha and the kernel upgraded to 5.10.52.
LF5.10.35_2.0.0	06/2021	Upgraded to 5.10.35 kernel.
LF5.10.9_1.0.0	03/2021	Upgraded to 5.10.9 kernel.
L5.4.70_2.3.0	01/2021	Updated the command lines in Section "Running the Arm Cortex-M4 image".
L5.4.70_2.3.0	12/2020	i.MX 5.4 consolidated GA for release i.MX boards including i.MX 8M Plus and i.MX 8DXL.

Revision history...*continued*

Revision number	Date	Substantive changes
L5.4.47_2.2.0	09/2020	i.MX 5.4 Beta2 release for i.MX 8M Plus, Beta for 8DXL, and consolidated GA for released i.MX boards.
L5.4.24_2.1.0	06/2020	i.MX 5.4 Beta release for i.MX 8M Plus, Alpha2 for 8DXL, and consolidated GA for released i.MX boards.
L5.4.3_2.0.0	04/2020	i.MX 5.4 Alpha release for i.MX 8M Plus and 8DXL EVK boards.
LF5.4.3_1.0.0	03/2020	i.MX 5.4 Kernel and Yocto Project Upgrades.
L4.19.35_1.1.0	10/2019	i.MX 4.19 Kernel and Yocto Project Upgrades.
L4.19.35_1.0.0	07/2019	i.MX 4.19 Beta Kernel and Yocto Project Upgrades.
L4.14.98_2.0.0_ga	04/2019	i.MX 4.14 Kernel upgrade and board updates.
L4.14.78_1.0.0_ga	01/2019	i.MX 6, i.MX 7, i.MX 8 family GA release.
L4.14.62_1.0.0_beta	11/2018	i.MX 4.14 Kernel Upgrade, Yocto Project Sumo upgrade.
L4.9.123_2.3.0_8mm	09/2018	i.MX 8M Mini GA release.
L4.9.88_2.2.0_8qxp-beta2	07/2018	i.MX 8QuadXPlus Beta2 release.
L4.9.88_2.1.0_8mm-alpha	06/2018	i.MX 8M Mini Alpha release.
L4.9.88_2.0.0-ga	05/2018	i.MX 7ULP and i.MX 8M Quad GA release.
L4.9.51_imx8mq-ga	03/2018	Added i.MX 8M Quad GA.
L4.9.51_8qm-beta2/8qxp-beta	02/2018	Added i.MX 8QuadMax Beta2 and i.MX 8QuadXPlus Beta.
L4.9.51_imx8mq-beta	12/2017	Added i.MX 8M Quad.
L4.9.51_imx8qm-beta1	12/2017	Added i.MX 8QuadMax.
L4.9.51_imx8qxp-alpha	11/2017	Initial release.

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

For more information, please visit: https://www.nxp.com

Date of release: 15 December 2023
Document identifier: IMXVPUAPI