

# i.MX VPU Application Programming Interface Linux® Reference Manual

## Contents

## 1 Overview

The i.MX SoC with Video Processing Units (VPU) support 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

1	Overview.....	1
2	i.MX 6 VPU Host Interface.....	6
3	i.MX 6 VPU API Features.....	8
4	i.MX 6 VPU Control.....	69
5	Revision History.....	87



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 `hantrade.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 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 state machine should be maintained through one event handler implemented on Arm.

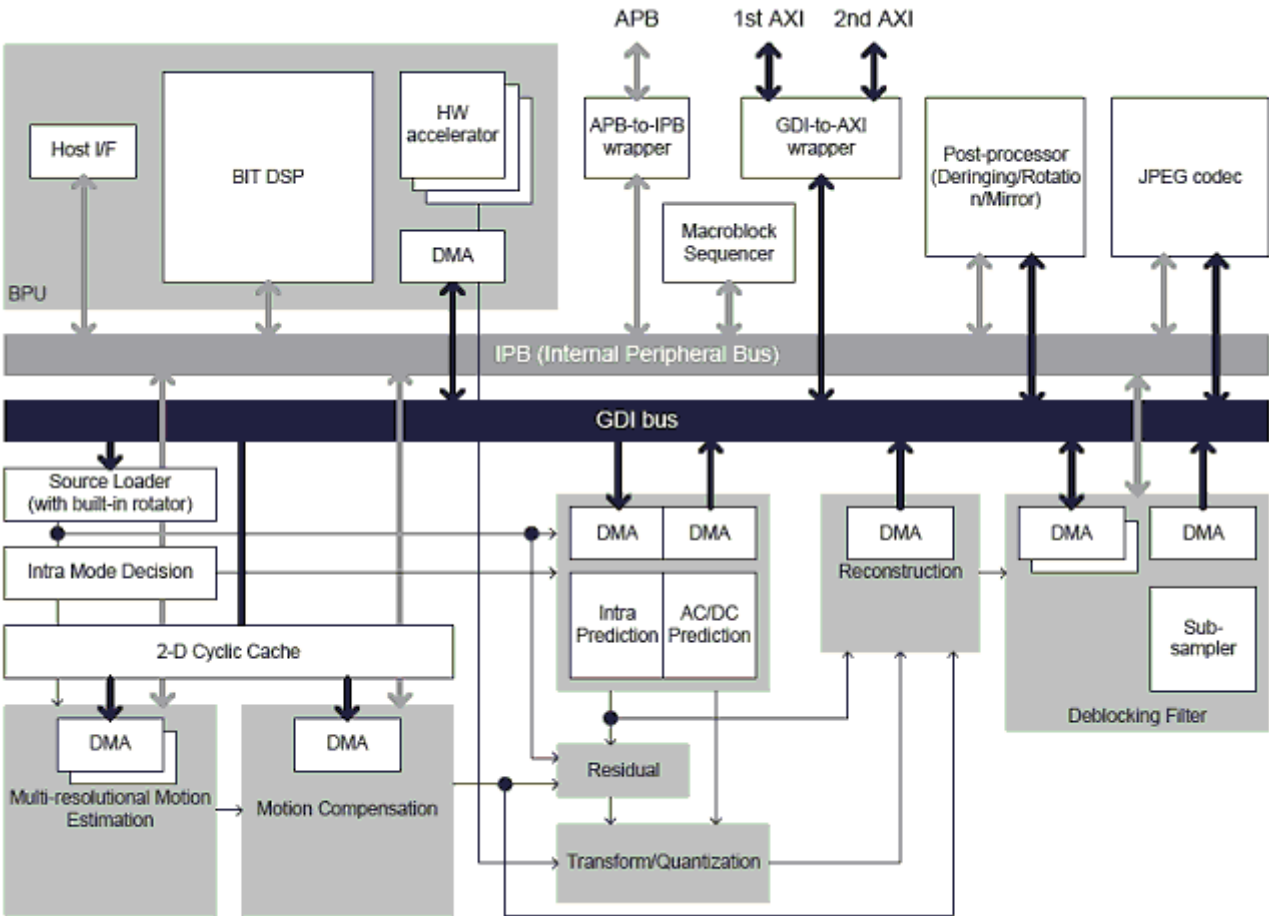
## 1.4 i.MX 6 VPU Overview

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

### 1.4.1 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  $[\pm 128, \pm 64]$
  - 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

## 1.4.2 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.

### 1.4.2.1 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.

### 1.4.2.2 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 Kbytes, this is not a large burden for the VPU operations in performance and memory bandwidth.

### 1.4.2.3 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 regarded 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.

## 2 i.MX 6 VPU Host Interface

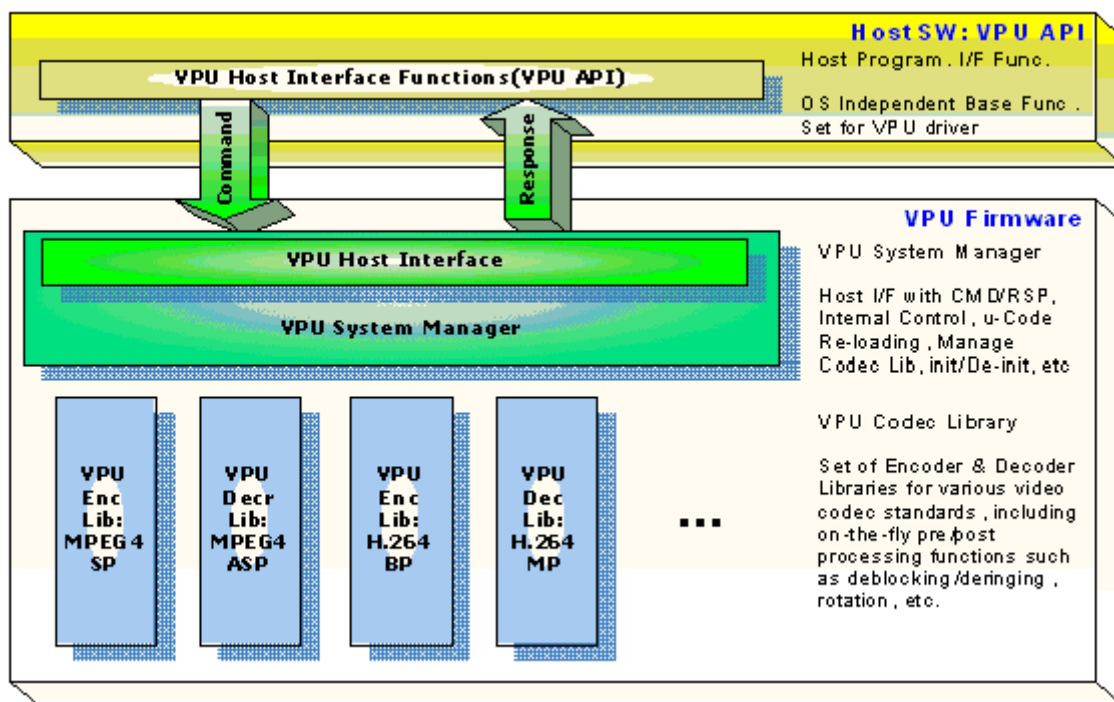
This section describes the interfaces used by host processor to control i.MX 6 VPU.

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

### 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 2. 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.

## 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.

## 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:

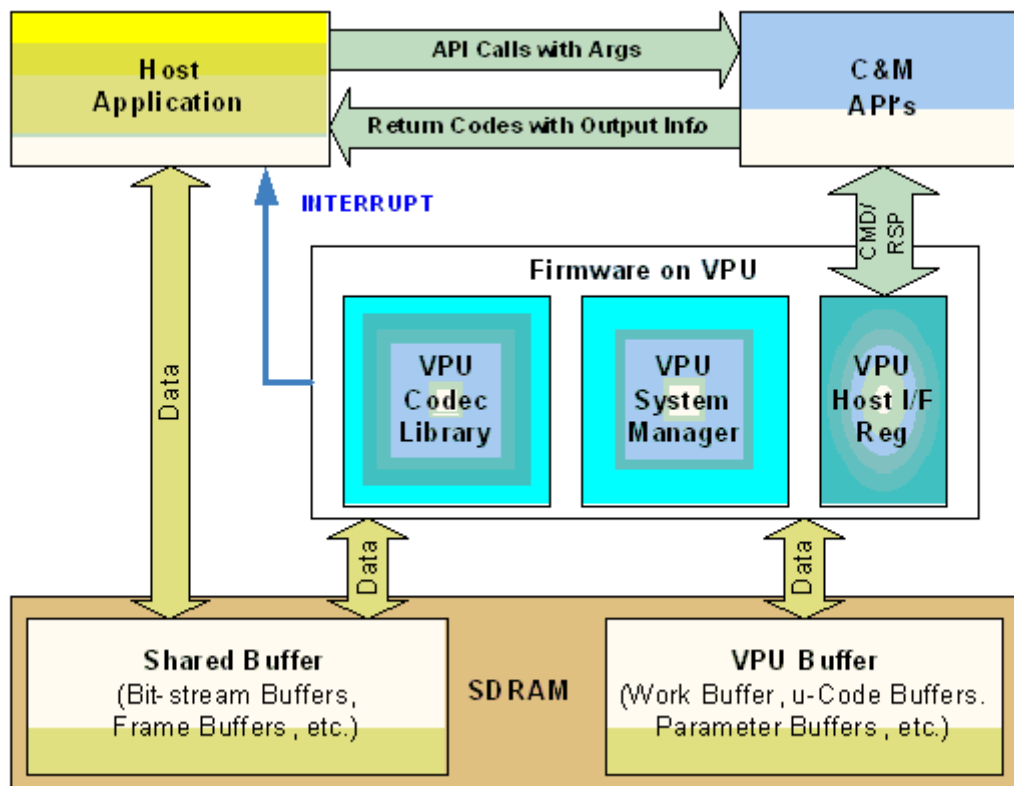
- BIT processor control registers update or show BIT processor status to host processors. Most of these registers are used for initializing BIT processor during boot-up.
- BIT processor global registers store all the global variables which are reserved even while an active instance is changed. All the 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.

## 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, signalling 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 3. Software Control Model of VPU from Host Application**

## 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.

### 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, `vpv_Init()`, and no additional information is required for calling this API. `vpv_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.

#### 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.

### 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.

## 3.2 Type Definitions

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

### 3.2.1 Type Definitions (common data types)

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

#### 3.2.1.1 Uint8

```
typedef unsigned char Uint8;
```

##### Description

8-bit unsigned integer type used for declaring pixel data.

#### 3.2.1.2 Uint16

```
typedef unsigned short Uint16;
```

##### Description

16-bit unsigned integer type.

#### 3.2.1.3 Uint32

```
typedef unsigned long Uint32;
```

##### Description

32-bit unsigned integer type used for declaring unsigned variables with wide ranges such as the size of a buffer.

#### 3.2.1.4 Uint64

```
typedef unsigned long long Uint64;
```

##### Description

64-bit unsigned integer type used for multiplication operations.

### 3.2.1.5 Int64

```
typedef long long Int64;
```

#### Description

64-bit signed integer type used for multiplication operations.

### 3.2.1.6 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.

### 3.2.1.7 VirtualAddress

```
typedef Uint32 VirtualAddress;
```

#### Description

Represents virtual addresses that are recognizable by CPU.

### 3.2.1.8 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.

**3.2.1.9 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

*Table continues on the next page...*

Table 1. Return Codes (continued)

Code	Description
RETCODE_FAILURE	Operation not successfully; this value is returned when an unrecoverable 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)
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 <b>DecRegisterFrameBuffer()</b> 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, <b>vpu_Init()</b> , 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 <b>vpu_DecStartOneFrame()</b> API calling.
RETCODE_JPEG_BIT_EMPTY	The filled data in the bitstream buffer is not enough for the header parser in the <b>vpu_DecStartOneFrame()</b> API calling.

### 3.2.1.10 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 [vpu\\_EncGiveCommand\(\)](#) and [vpu\\_DecGiveCommand\(\)](#).

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

```

### 3.2.1.11 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.

### 3.2.1.12 MirrorDirection

```

typedef enum {

    MIRROR_NONE,

    MIRROR_VER,

    MIRROR_HOR,

    MIRROR_HOR_VER

} MirrorDirection;

```

**Description**

Enumeration type for representing the mirroring direction.

**3.2.1.13 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.

**3.2.1.14 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.

**3.2.1.15 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.

**3.2.1.16 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.

## 3.2.2 Data and Structure Definitions

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

### 3.2.2.1 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.

### 3.2.2.2 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.

### 3.2.2.3 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.

### 3.2.2.4 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

## i.MX 6 VPU API Features

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.

### 3.2.2.5 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.

### 3.2.2.6 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.

### 3.2.2.7 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

### 3.2.2.8 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.

**3.2.2.9 EncMjpgParam**

```

typedef struct {

    int mjpg_sourceFormat;

    int mjpg_restartInterval;

    int mjpg_thumbNailEnable;

    int mjpg_thumbNailWidth;

```

```

    int mjpg_thumbNailHeight;

    Uint8 * mjpg_hufTable;

    Uint8 * 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

### 3.2.2.10 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.

**3.2.2.11 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 MESSearchRange;

    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  $(\text{numerator}/(\text{denominator} + 1))$ , which equals  $(\text{frameRateInfo} \& 0xffff) / ((\text{frameRateInfo} \gg 16) + 1)$ . 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 [EncMjpgParam](#).

`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.

## i.MX 6 VPU API Features

`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\_MJPEG.

`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\_MJPEG.

`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\_MJPEG.

`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.

`mjpegParam` 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 `userQpMin` is in the range of 1 to 31 and less than `userQpMax`. In H.264 mode, the value of `userQpMin` 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  $\text{factor} \times 32768$ , 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  $0.75 \times 32768$ .

`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  $2 \times (720/16) = 90$ . 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.

### 3.2.2.12 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.

### 3.2.2.13 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.

### 3.2.2.14 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).

### 3.2.2.15 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.

### 3.2.2.16 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.

### 3.2.2.17 SearchRamParam

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

#### Description

This is not used in the i.MX 6.

### 3.2.2.18 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.

### 3.2.2.19 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 [CodStd](#).

`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.

### 3.2.2.20 DecReportBufSize

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

#### Description

Not used in the i.MX 6.

### 3.2.2.21 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:  $\text{picBufWidth} = ((\text{picWidth} + 15)/16) \times 16$ , 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:  $\text{picBufHeight} = ((\text{picHeight} + 15)/16) \times 16$ , 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 `vpv_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.

### 3.2.2.22 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.

### 3.2.2.23 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.

### 3.2.2.24 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)<sup>th</sup> 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.

### 3.2.2.25 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.

### 3.2.2.26 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.

### 3.2.2.27 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 altrf (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.

### 3.2.2.28 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.

### 3.2.2.29 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.

### 3.2.2.30 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 `vpv_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 occurs 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.

## i.MX 6 VPU API Features

`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,  $\text{frame rate} = \text{frameRateRes} / (\text{frameRateDiv} * 2)$ . Otherwise,  $\text{frame rate} = \text{frameRateRes} / \text{frameRateDiv}$ .

`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.

### 3.2.2.31 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.

**3.2.2.32 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.

**3.2.2.33 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.

**3.3 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.

### 3.3.1 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.

#### 3.3.1.1 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  $n^{\text{th}}$  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.

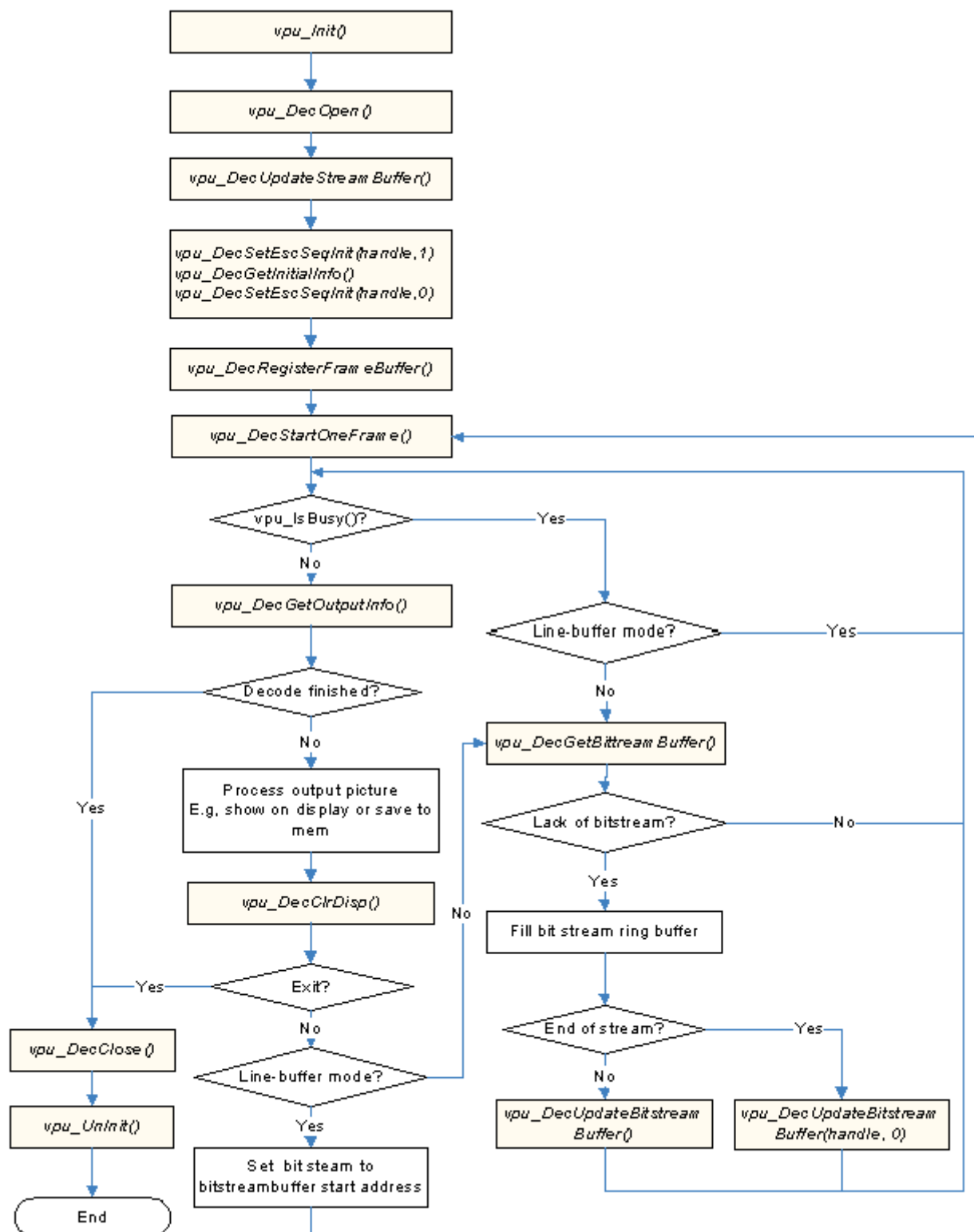


Figure 4. Decoder Operation Flow

### 3.3.1.2 Encoder Operation Flow

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

## i.MX 6 VPU API Features

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.

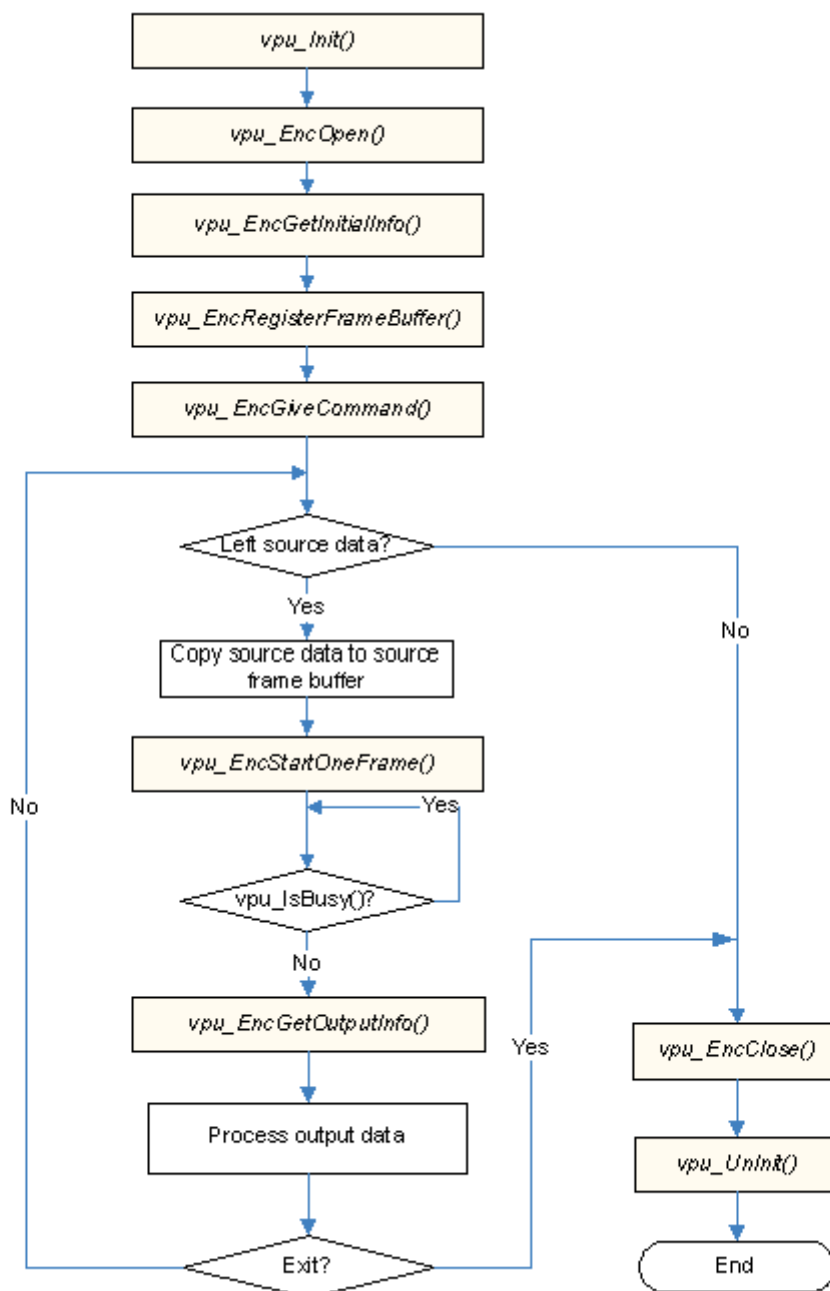


Figure 5. Encoder Operation Flow

## 3.3.2 Control API

The following sections describe the control API functions.

### 3.3.2.1 vpu\_Init()

Prototype

## i.MX 6 VPU API Features

```
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.

## 3.3.2.2 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.

## 3.3.2.3 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.

## 3.3.2.4 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.

### 3.3.2.5 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`.

### 3.3.2.6 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.

### 3.3.2.7 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.

### 3.3.2.8 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.

### 3.3.2.9 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.

### 3.3.2.10 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.

### 3.3.2.11 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.

### 3.3.2.12 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*.

## 3.3.3 Encoder API

The following sections describe the encoder API functions.

### 3.3.3.1 vpu\_EncOpen()

#### Prototype

## i.MX 6 VPU API Features

```
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.

## 3.3.3.2 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.

### 3.3.3.3 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.

### 3.3.3.4 vpu\_EncGetBitstreamBuffer()

#### Prototype

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

#### Parameter

handle [input] is an encoder handle obtained from **vpu\_EncOpen()**.

prdPtr [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, prdPtr, 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.

### 3.3.3.5 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.

### 3.3.3.6 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()**.

### 3.3.3.7 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()**.

### 3.3.3.8 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.

### 3.3.3.9 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 CodecCommand type.

param [input/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	<p>handle is a pointer to MirrorDirection. *param should be one of the following:</p> <ul style="list-style-type: none"> <li>MIRDIR_NONE means no mirroring.</li> <li>MIRDIR_VER means vertical mirroring.</li> <li>MIRDIR_HOR means horizontal mirroring.</li> <li>MIRDIR_HOR_VER means both directions mirroring.</li> </ul> <p>Return values are as follows:</p> <p>RETCODE_SUCCESS means that the given mirroring direction is valid.</p> <p>RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given mirroring direction is invalid.</p>
SET_ROTATION_ANGLE	<p>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:</p> <p>RETCODE_SUCCESS means that the given rotation angle is valid.</p> <p>RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given rotation angle is invalid.</p> <p><b>NOTE:</b> Rotation angle cannot be changed after sequence initialization since it might cause problems in handling frame buffers.</p>

*Table continues on the next page...*

**Table 3. Encoder Commands (continued)**

Command	Description
ENC_GET_SPS_RBSP	<p>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:</p> <p>RETCODE_SUCCESS SPS means that it is successfully generated and available at the received buffer pointer.</p> <p>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.</p> <p>RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer or contains improper values for some member variables.</p>
ENC_GET_PPS_RBSP	<p>param is a pointer to an EncParamSet type structure. Return values are as follows:</p> <p>RETCODE_SUCCESS PPS means that it is successfully generated and available at the received buffer pointer.</p> <p>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.</p> <p>RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer or contains improper values for some member variables.</p>
ENC_PUT_MP4_HEADER	<p>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:</p> <p>RETCODE_SUCCESS means that the requested header syntax is successfully inserted.</p> <p>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.</p> <p>RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer or contains improper values for some member variables.</p>
ENC_PUT_AVC_HEADER	<p>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:</p> <p>RETCODE_SUCCESS means that the requested header syntax successfully inserted.</p> <p>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.</p> <p>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</p>
ENC_SET_SEARCHRAM_PARAM	The command is not used in i.MX 6.
ENC_SET_INTRA_MB_REFRESH_NUMBER	<p>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:</p> <p>RETCODE_SUCCESS means that the requested header syntax is successfully inserted,</p>
ENC_ENABLE_HEC	<p>param is ignored. Return values are as follows:</p> <p>RETCODE_SUCCESS means that the requested header syntax is successfully inserted.</p>

*Table continues on the next page...*

**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 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 headerType, 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.  RETCODE_INVALID_PARAM means that the given argument parameter, param or headerType, 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 headerType, 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 headerType, 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.

*Table continues on the next page...*

**Table 3. Encoder Commands (continued)**

Command	Description
	RETCODE_INVALID_PARAM means that the given argument parameter, param or headerType, 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_SLICEINFO	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 <b>vpu_EncGetInitialInfo()</b> . 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. <ul style="list-style-type: none"> <li>• 0 - random intra refresh mode</li> <li>• 1 - consecutive intra refresh mode</li> </ul>
ENC_ENABLE_SOF_STUFF	Pad stuffing zero bytes to the end of JPEG SOF fields. <ul style="list-style-type: none"> <li>• 0 - disable stuffing</li> <li>• 1 - enable stuffing</li> </ul>

### 3.3.4 Decoder API

The following sections describe the decoder API functions.

#### 3.3.4.1 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.

### 3.3.4.2 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.

### 3.3.4.3 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 [DecInitialInfo](#). 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()**.

### **3.3.4.4 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.

### **3.3.4.5 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.

**3.3.4.6 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.

### 3.3.4.7 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()**.

### 3.3.4.8 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.

## 3.3.4.9 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.

### 3.3.4.10 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.

### 3.3.4.11 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\_MJPEG codec.

## 3.3.4.12 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 CodecCommand 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: <ul style="list-style-type: none"> <li>MIRDIR_NONE-No mirroring</li> <li>MIRDIR_VER-Vertical mirroring</li> <li>MIRDIR_HOR-Horizontal mirroring</li> <li>MIRDIR_HOR_VER-Both directions</li> </ul>

*Table continues on the next page...*

**Table 4. Decoder Commands (continued)**

Command	Description
	<p>Return values are as follows:</p> <p>RETCODE_SUCCESS means that the given mirroring direction is valid.</p> <p>RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given mirroring direction is invalid.</p>
SET_ROTATION_ANGLE	<p>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:</p> <p>RETCODE_SUCCESS means that the given rotation angle is valid.</p> <p>RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given rotation angle is invalid.</p>
SET_ROTATOR_OUTPUT	<p>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:</p> <p>RETCODE_SUCCESS means that the given frame buffer pointer is valid.</p> <p>RETCODE_INVALID_PARAM means that the given argument parameter, param, is invalid so given frame buffer pointer is invalid.</p>
SET_ROTATOR_STRIDE	<p>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:</p> <p>RETCODE_SUCCESS means that the given stride value is valid.</p> <p>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.</p>
DEC_SET_SPS_RBSP	<p>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 DecParamSet 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:</p> <p>RETCODE_SUCCESS means that transferring a SPS RBSP to a decoder completed successfully.</p> <p>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.</p> <p>RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer or contains improper values for some member variables.</p>
DEC_SET_PPS_RBSP	<p>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 DecParamSet 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:</p> <p>RETCODE_SUCCESS means that transferring a PPS RBSP to decoder completed successfully.</p> <p>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.</p> <p>RETCODE_INVALID_PARAM means that the given argument, param, is invalid. It has a null pointer, or contains improper values for some member variables.</p>
ENABLE_DERING	<p>Enables VPU internal dering operation. Returns RETCODE_SUCCESS.</p>

*Table continues on the next page...*

**Table 4. Decoder Commands (continued)**

Command	Description
DISABLE_DERING	Disables VPU internal dering function. Returns RETCODE_SUCCESS.
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 <b>vpu_DecGetInitialInfo()</b> . 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 <b>vpu_DecGetInitialInfo()</b> . 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 <b>vpu_DecGetInitialInfo()</b> . 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 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.

## 4.1 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.

### 4.1.1 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.

### 4.1.2 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.

### 4.1.3 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 **vpv\_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 **vpv\_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.

#### 4.1.4 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.

#### 4.1.5 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.

##### 4.1.5.1 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, **vpv\_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 **vpv\_DecUpdateBitStreamBuffer()**.

## 4.1.6 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.

## 4.2 Encoder Control

### 4.2.1 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 [vpu\\_EncGiveCommand\(\)](#).

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.

## 4.2.2 Configuring VPU for Encoder Instance

### 4.2.2.1 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.

### 4.2.2.2 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.

### 4.2.2.3 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 **vp\_u\_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 **vp\_u\_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.

## 4.2.3 Running Picture Encoder on VPU

### 4.2.3.1 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.

### 4.2.3.2 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, **vp\_u\_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.

### 4.2.3.3 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.

### 4.2.3.4 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.

### 4.2.3.5 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 6SoloLite Applications Processor Reference Manual (IMX6SLRM)*
- *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 an 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, **vpv\_EncGetOutputInfo()**, the next picture encoding operation is not initiated by calling **vpv\_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 **vpv\_EncGetOutputInfo()** function as a releasing command of the VPU from the current picture encoding operation.

### 4.2.4 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 **vpv\_EncClose()** function.

### 4.2.5 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.

## 4.3 Decoder Control

### 4.3.1 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 a 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.

#### 4.3.1.1 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  $\text{MAX}(\text{reference frame number}, 16) + 2$  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.

### 4.3.2 Configuring VPU for Decoder Instance

### 4.3.2.1 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, `vpv_DecGetBitstreamBuffer()`. Updating the write pointer is accomplished using the API function, `vpv_DecUpdateBitstreamBuffer()`.

### 4.3.2.2 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<sup>th</sup> 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, **vpv\_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 **vpv\_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 **vpv\_SetSeqInitEsc()** function again. This flag affects all the decoder instances performing a DEC\_SEQ\_INIT operation.

### 4.3.2.3 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 **vpv\_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 **vpv\_DecGetInitialInfo()**.

## 4.3.3 Running Picture Decoder On VPU

### 4.3.3.1 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 recommended 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<sup>th</sup> 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.

### 4.3.3.2 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 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.

### 4.3.3.3 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)<sup>th</sup> 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.

#### 4.3.3.4 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.

#### 4.3.3.5 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.

### 4.3.3.6 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, **vpv\_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, **vpv\_DecGetOutputInfo()**, the next picture decoding operation is not initiated by calling **vpv\_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 **vpv\_DecGetOutputInfo()** function as a releasing command of the VPU from the current picture decoding operation.

#### 4.3.3.6.1 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<sup>th</sup> 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.

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.

#### 4.3.3.6.2 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 **vpv\_DecStartOneFrame()** after clearing the display flag by calling **vpv\_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.

#### 4.3.3.6.3 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.

#### 4.3.3.6.4 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.

#### 4.3.3.6.5 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.

#### 4.3.3.6.6 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.

#### 4.3.3.7 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()`.

### 4.3.4 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.

## 4.3.5 Terminating a Decoder Instance

### 4.3.5.1 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 `vpv_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 `vpv_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 `vpv_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.

### 4.3.5.2 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 `vpv_DecClose()` function.

## 4.3.6 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

## 4.4 Example Applications

This section discusses the example applications provided for i.MX 6 VPU API.

### 4.4.1 VPU Library

VPU library and header file source code is located under Yocto Project build tmp work tree in `imx-lib/*/vpv`. 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 `vpv_fw_imx6q.bin` or `vpv_fw_imx6d.bin` file is located. If this variable is not exported by the user, the `vpv_fw_mx6.bin` file must be located in the `/lib/firmware/vpv` directory.

## 4.4.2 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. [Decode Stream to Display on LCD](#), and [Encode Stream from Camera Captured Data](#), 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.

### 4.4.2.1 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  $n^{\text{th}}$  frame buffer, clear the buffer display flag using **vpu\_DecClrDispFlag()**. This function does not need to be called for the STD\_MJPEG 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.

#### 4.4.2.2 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.

### 4.4.3 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 Revision History

This table provides the revision history.

**Table 6. Revision history**

Revision number	Date	Substantive changes
L4.9.51_imx8qxp-alpha	11/2017	Initial release
L4.9.51_imx8qm-beta1	12/2017	Added i.MX 8QuadMax
L4.9.51_imx8mq-beta	12/2017	Added i.MX 8M Quad
L4.9.51_8qm-beta2/8qxp-beta	02/2018	Added i.MX 8QuadMax Beta2 and i.MX 8QuadXPlus Beta
L4.9.51_imx8mq-ga	03/2018	Added i.MX 8M Quad GA
L4.9.88_2.0.0-ga	05/2018	i.MX 7ULP and i.MX 8M Quad GA release
L4.9.88_2.1.0_8mm-alpha	06/2018	i.MX 8M Mini Alpha release
L4.9.88_2.2.0_8qxp-beta2	07/2018	i.MX 8QuadXPlus Beta2 release
L4.9.123_2.3.0_8mm	09/2018	i.MX 8M Mini GA release
L4.14.62_1.0.0_beta	11/2018	i.MX 4.14 Kernel Upgrade, Yocto Project Sumo upgrade
L4.14.78_1.0.0_ga	01/2019	i.MX6, i.MX7, i.MX8 family GA release

*Table continues on the next page...*

**Table 6. Revision history (continued)**

Revision number	Date	Substantive changes
L4.14.98_2.0.0_ga	04/2019	i.MX 4.14 Kernel upgrade and board updates



**How to Reach Us:****Home Page:**[nxp.com](http://nxp.com)**Web Support:**[nxp.com/support](http://nxp.com/support)

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: [nxp.com/SalesTermsandConditions](http://nxp.com/SalesTermsandConditions).

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro,  $\mu$ Vision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2019 NXP B.V.

Document Number IMXVPUAPI  
Revision L4.14.98-2.0.0\_ga, 04/2019

