# The User Guide of Video Codec Engine Library

| | |
|---|---|
| Author | CXC |
| Date Created | 2011.3.29 |
| Current Version | 1.01 |

Version History:

| Date | Vesion | Modify | Contents |
|---|---|---|---|
| | | | |
| 2011.3.29 | 1.00 | Calder | Create this document. |

# Content

# Preface

The Video Codec Engine is an embedded logic circuit in our Chip. This circuit consists of decoders such as MPEG1/2/4, JPEG, H.263, H.264, AVS, VC-1, WMV7/8 and VP-6. It also contains an H.264 video encoder. So far based on this engine, we have developed a library to decode video streams using the video codec engine, this library is called 'libve'.

The Libve is designed to work with different players on Linux or other operating systems. With this library, you can play video streams, and easily implement some tricks such as jump play, fast forward, fast backward, rotate videos, scaledown videos, get video preview. This document describes the user interface of Libve, and also introduce an architecture of video decode driver base on the Libve.

To link and use the libve, you should get the 'libve.a' file and three C code header files, libve.h, libve_adapter.h and libve_typedef.h.

To show how the libve works, we give a demo. This demo use the 'VBV' module described in 'vbv.h' and 'vbv.c' to receive video bitstream data and manage video bitstream frames. When libve_decode method is called, libve will request bitstream frames from the 'VBV' module for decoding. This demo also use the 'FBM' module described in 'fbm.h' and 'fbm.c' to manage picture frames. Libve requests empty frame buffer from the 'FBM' module, decode one picture to the buffer and then returns it to the 'FBM' module. The control flow of using 'VBV', 'FBM' modules and libve to decode is described in a C code file 'vdecoder.c'.

# Glossary

| | |
|---|---|
| FBM | Frame buffer manager, a program module which is responsible for frame buffer management. |
| ISR | Interrupt service routine. |
| Interface | A set of methods for a specific functionality. |
| OS | Operating system, such as ‘MELIS’, ‘LINUS’ and ‘Android’. |
| PCR | Program clock reference (a concept described in MPEG-2). |
| PTS | Presentation time stamp, each video frame has a PTS to tell when it should be displayed. |
| VBV | Video bitstream verification (a concept described in MPEG-2), a program module which is responsible for video bitstream management. |
| VE | The video codec engine embedded in our chip. |

# Reference

# PART A Requirement

## A.1. Application Scene

The Libve is expected to be used in several applications on MELIS or LINUX, based on our chips. Application on these platforms may be CEDAR player, MPlayer, CEDAR-X player or Video Previewer. In these applications, a driver uses libve to decode video stream and output pictures.

Before decoding, the driver opens the libve with configure information and video stream information. The driver may have a thread to control the video decoding process. It gets bitstream frame from a VBV module and feed it to the library. The libve will create one or two FBM to store pictures. When need picture to display, the driver can get a FBM handle from the library and request pictures from the FBM. To quit the decoding process, the driver should close the library. When some error happens or when the driver wants the library to clear its decoding status, the driver may call the the library's reset function.

## A.2. Function Requirement

## A.3. Supplement

# PART B Architecture

## B.1. Architecture of Libve

As showed in figure 1, the library is devided into two parts, the 'top level control part' and the 'specific decoders'. The top level control part is responsible of some common settings. It uses the decoder interface offered by specific decoder modules to decode stream, and provides an interface for outside program to control the library. The specific decoder level consists of H.264 decoder, MPEG-2 decoder, MPEG-4 decoder, VC-1 decoder and MJPEG decoder. These decoders provide an interface for the top level control module. They only do the pure decoding work and take no care of bitstream and frame buffer management. The two modules share some global configurations like VE version, maximum VE speed and etc. Also some global methods are shared, mainly for register accessing, error reporting and ve reset.

The library uses an Adapter module to make itself be Application independent. It use four interfacies offered by the adapter to access resource which are application dependent. The IVE interface contains methods for VE controlling, such as enable or disable clock for VE, reset VE, enable VE interrupt, etc. The IFBM interface contains methods for controlling frame buffer manager, the library use this interface to request or release frame buffers for decoding. The IVBV interface contains methods for video bitstream accessing, the library use this interface to request or release video bitstream frames for decoding. The IOS interface contains methods which should be offered by the OS, such as heap operations (malloc, palloc), memory operations (memset, memcpy), printing operations (printf), etc. When a program wants to link the library, it should implement these three interfacies for the library.

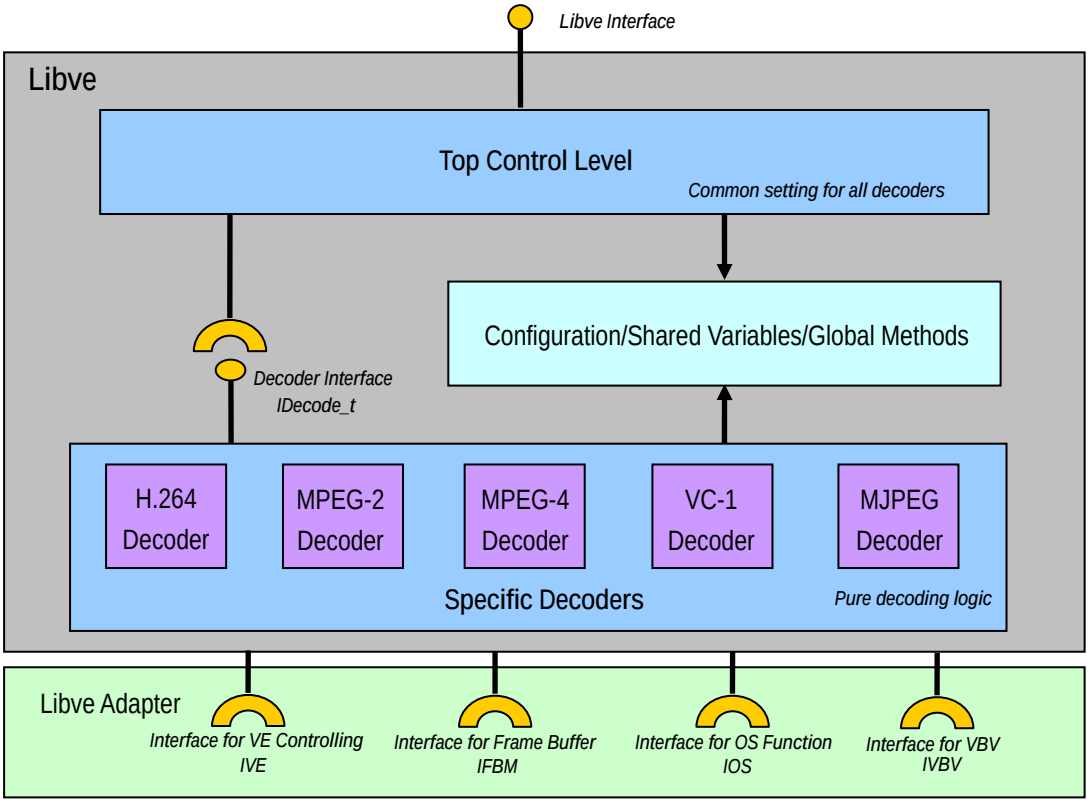


Figure 1

## B.2. Recommended Architecture of Drivers using Libve

As showed in Figure 2, it is recommended that a driver engineer use three modules to implement a driver for video decoding, the VBV module, the Libve module and the FBM module. The VBV module is a bitstream FIFO, which is responsible of receiving stream data and providing stream frame. The FBM module is a frame buffer manager, responsible of managing empty frame buffer and display queue. The library module provides decoding functions, and is also responsible for picture reordering, picture rotation and picture scale down.

In this architecture, application can choose any implementation solution for the VBV or FBM module, without concerning with the video decoding issue, as long as the FBM module maintains an IFBM interface for libve. As the driver use an OS adapter module to make itself be OS independent, most of the code can be reused when porting the driver between different operating systems. As all modules except the libve are hardware independent, the driver is also easy to be porting between different chips. Just change the 'libve.a' file for specific chip version and compile the driver again.

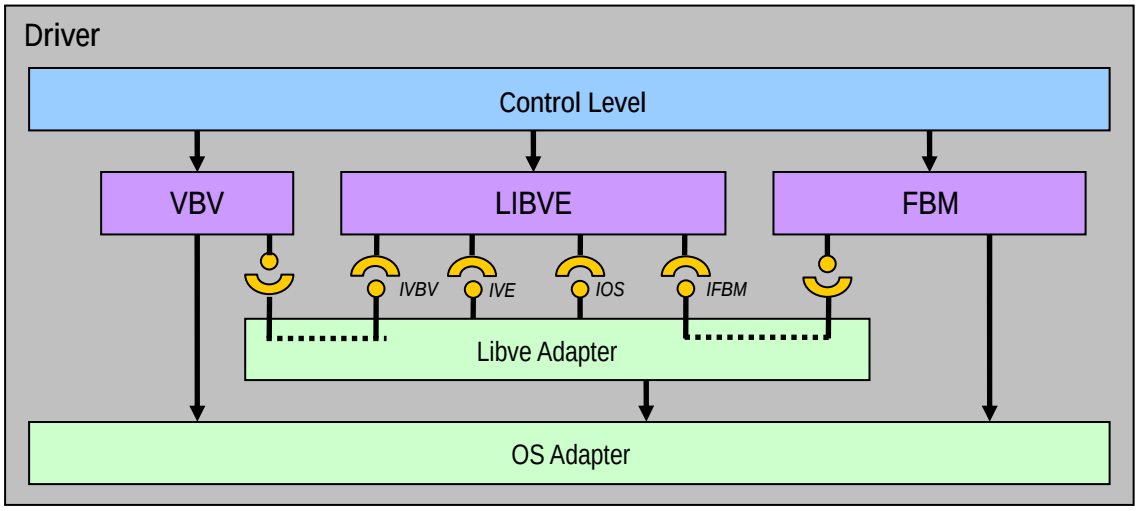


Figure 2

Refer to the annex part for the recommended implementation of VBV and FBM module.

# B.3. Basic Execute Flows

## B3.1. Open Libve

The library should be opened before being used to decode streams. User calls the method 'libve_open' with configuration information and stream information to open the library. This action may trigger the library to initialize the FBM module using IFBM interface. If the specific decoder does not know the picture size yet, the FBM initialization may be left to later steps when decoding stream.

The specific decoder may allocate not only one FBM module. In case of scale down or rotation fuction is selected, the specific decoder may allocate two FBM modules, one with less frames for storing original pictures, the other with more frames for storing scaled or rotated pictures for displaying.

After 'libve_open' method is successfully returned, 'libve_set_vbv' method should be called before decoding bitstream. The library use the vbv handle to access and return video bitstream frames.

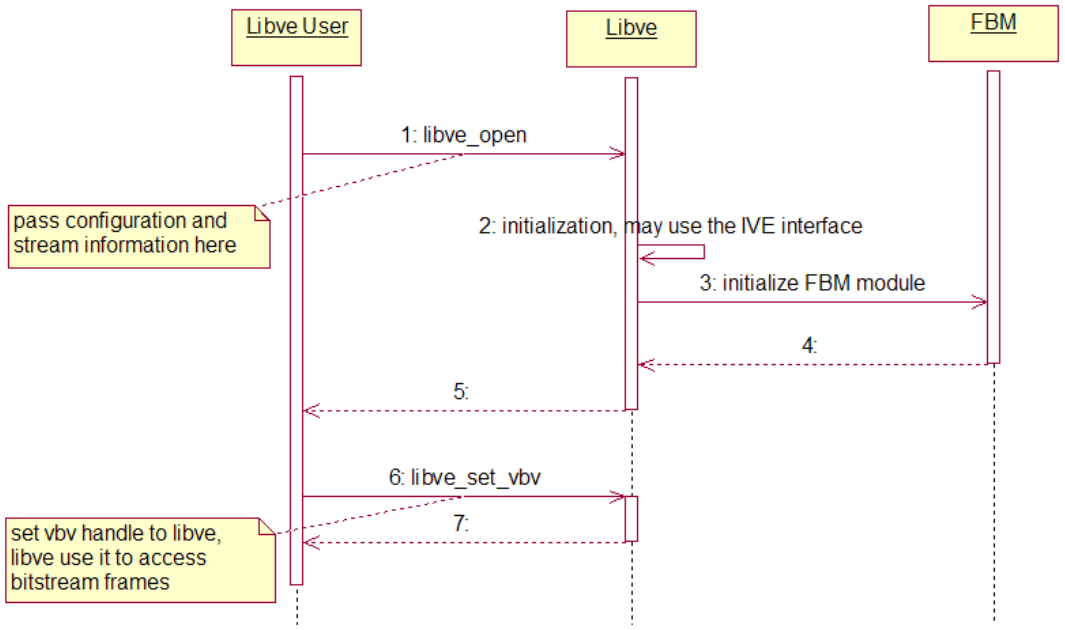


Figure 3

## B3.2. Close Libve

When user wants to quit, 'libve_close' method should be called to close the library. In this method, the bitstream frames oqupied by the library are returned to the VBV module throuth the IVBV interface. Picture frame buffers are also returned to the FBM module using IFBM interface. After that, the library will call the release method in IFBM module to release the FBM.

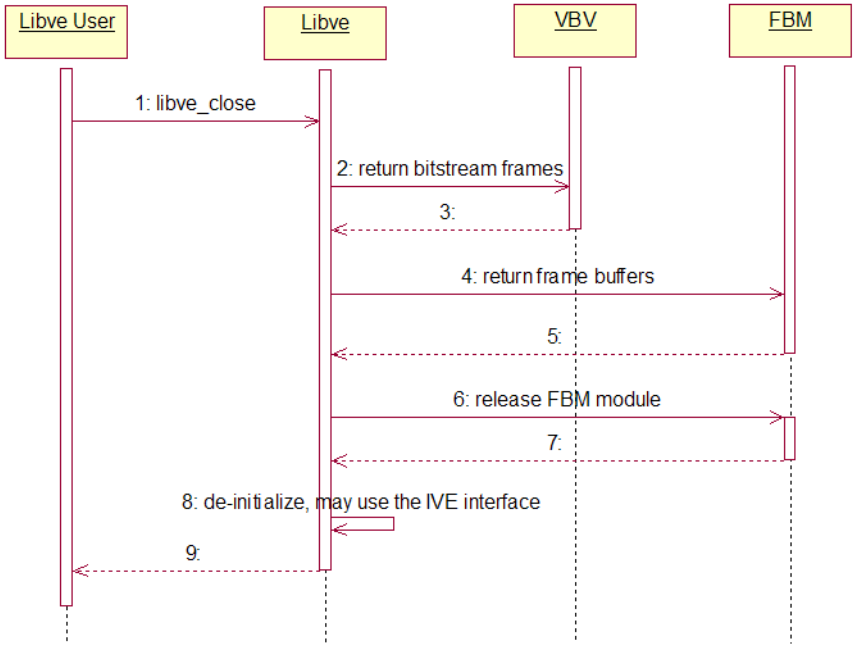


Figure 4

## B3.3. Decode Stream Frame

The library's user calls 'libve_decode' method to decode video pictures from bitstream. The library will request bitstream frame from the IVBV interface, request picture frame buffer from the IFBM interfece. After decoded, the bitstream frame is return to the VBV module through IVBV interface. Any time when a video picture is ready to output, the library will call the IFBM's return method or share method to output the video picture.

If the library request bitstream frame fails, 'libve_decode' method will return a code to tell no bitstream frame. If the library request frame buffer fails, 'libve_decode' will return a code to tell no empty picture frame buffer. If one frame is decoded, the 'libve_decode' will return a code to tell a common frame or a key frame is decoded.

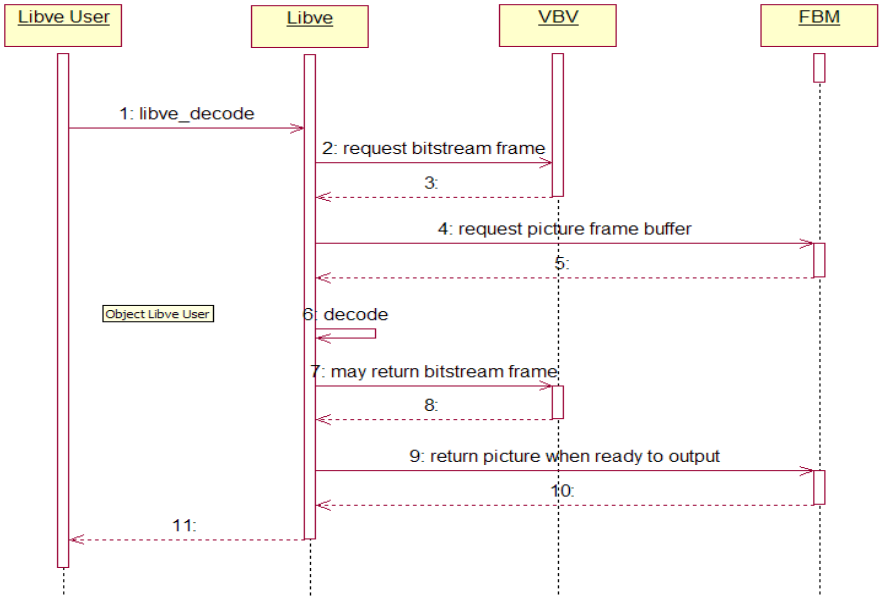

Figure 5

## B3.4. Reset Libve

When some error happens or video decoding process need be restarted, user can use 'libve_reset' method to reset the library. User can tell whether to put the pictures kept by the library to FBM's display queue, or just return them as invalid frames. The library also return bitstream frames kept inside to the VBV module. After reset, the library still keeps the initialize setting for the video stream. User can continue to call the decode method, and the specific decoder will choose a key frame to restart the decoding process.

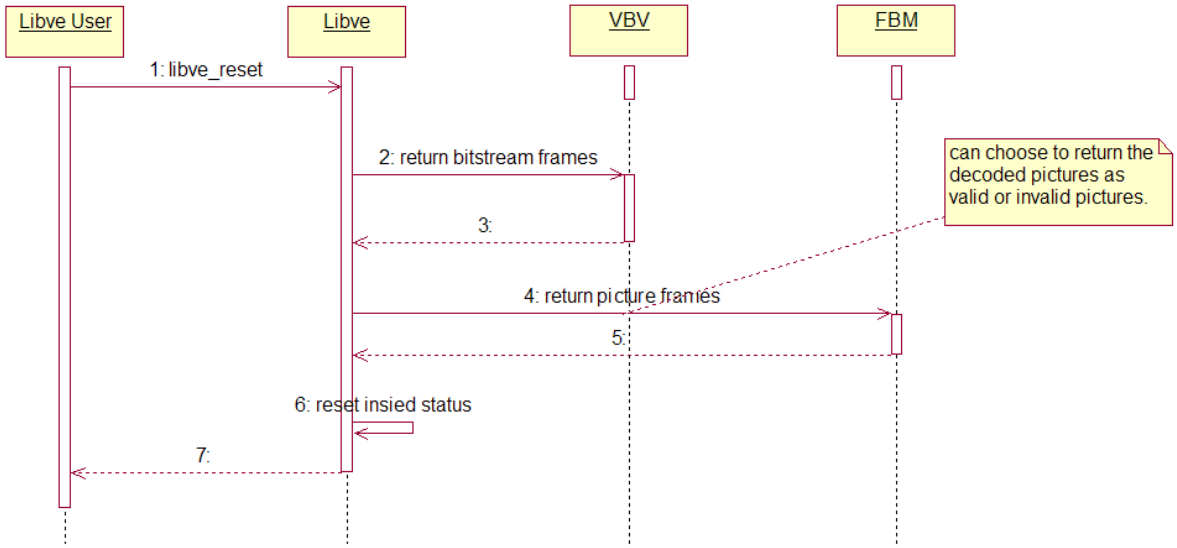


Figure 6

## B3.5. Request Pictures to Show and return Pictures

When the user wants to get a frame for display, it should get a handle of the FBM instance first. In case of scale down or rotation function is opened, the specific decoder may maintain two FBM instance, one for storing original pictures and the other for storing scaled or rotated pictures. When requested to give a handle of a FBM instance, it will give the one which is for display (the one storing scaled or rotated pictures).

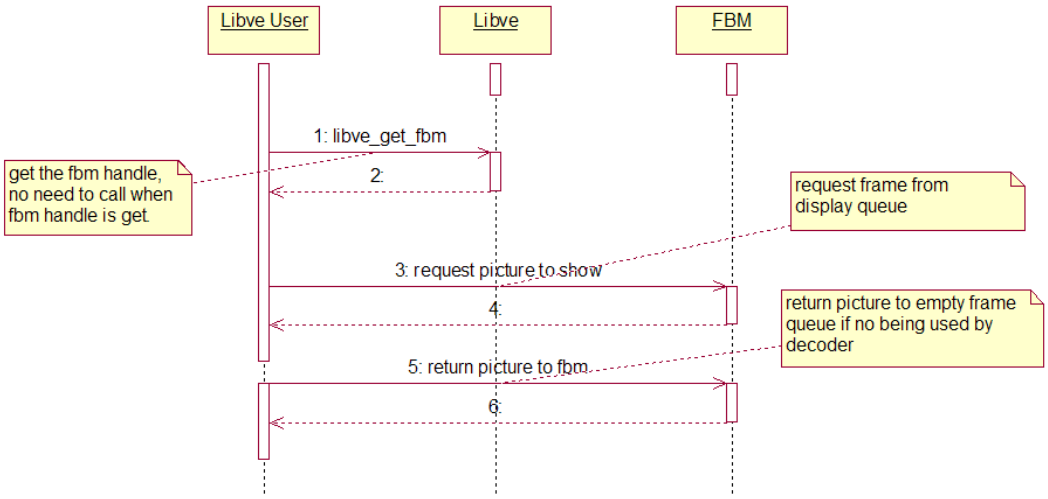


Figure 7

# PART C Interface Specification

This part describes methods the libve provides to its user, and methods provided to the libve (IVE, IFBM, IVBV and IOS).

Secsion C.1 introduces methods which are implemented by the library and provided to its user. The library users use this simple interface to decode video streams. Section C.2 describes three interfacies needed by the library for memory operation, frame buffer accessing, bitstream frame accessing and hardware controlling. These interfacies are designed to make the library be application independent.

## C.1. Libve Output Interface Design

The libve implements a simple interface for its user to decode video streams. This interface is declared in a header file named 'libve.h'. There are ten methods in this interface, as below:

Table-1 methods of libve output interface

| |
|---|
| Handle libve_open (vconfig_t* config, vstream_info_t* stream_info); |
| vresult_e libve_close (u8 flush_pictures, Handle ve); |
| vresult_e libve_reset (u8 flush_pictures, Handle ve); |
| vresult_e libve_set_vbv (Handle vbv, Handle ve); |
| Handle libve_get_fbm (Handle ve); |
| vresult_e libve_decode (u8 keyframe_only, u8 skip_bframe, u32 cur_time, Handle ve); |
| vresult_e libve_get_stream_info (vstream_info_t* vstream_info, Handle ve); |
| u8* libve_get_version (Handle ve); |
| u8* libve_get_last_error (Handle ve); |
| vresult_e libve_flush (u8 flush_pictures, Handle ve); |

Users call 'libve_open' to start the libve, pasing the configuration and video stream information. When Users want to quit, 'libve_close' should be called to release resource and clear hardware status. Users can call 'libve_reset'

to make the library restart the decoding process. 'libve_decode' is a method for decoding one frame of video stream data, before this method is called, users should use 'libve_set_vbv' to tell the library the VBV loop buffer's base address and its size. As the FBM module is initialized and maintained by the decoder, user need to call 'libve_get_fbm' to get the handle of the FBM instance to get decoded frames. Stream information may not be known by user before the stream is decoded, if so, the user can call 'libve_get_stream_info' to get information about the decoded video stream. 'libve_get_version' tells the current version of the library being used, and 'libve_get_last_error' gives a texture error description about the last operation.

Following we will describe each function of this interface.

## libve_open

'libve_open' is a method for library users to config and startup VE. When calling 'libve_open', users should pass in two parameters to tell the configuration and stream information. The first parameter 'config' is a pointer to a variable of vconfig_t structure type, as mentioned in table 3. Library users can config the library by this parameter. The second parameter 'stream_info' is a pointer to a variable of vstream_info_t structure type, as mentioned in table 4. The library uses this parameter to select a specific decoder and use it to startup the specific decoder.

When opening a specific decoder, the FBM module will be initialized if picture size is available by the decoder. If picture size is unknown, decoder will initialize the FBM module in 'libve_decode' method when picture size is decoded from bitstream.

In this method, the VE interrupt is enabled.

Table-2 libve_open

| | |
|---|---|
| Name | libve_open |
| Prototype | Handle libe_open (vconfig_t* config, vstream_info_t* stream_info); |
| Function | Start up the library. |
| Return | A handle of the VE device. |
| Input | vconfig_t* config, the configuration for the library, to set the maximum supported picture size, scale down ratio, rotation angle. |
| Input | vstream_info_t* stream_info, to tell the video stream information such as stream format, picture size, frame rate, etc. The 'stream_info' may contain some private data for a specific decoder. |
| Precondictions | |
| Success Guarantee | • Configuration and stream information are backuped by specific decoder;<br>• VE interrupt is enable;<br>• VE clock is opened; |
| Failed End Condition | • Parameter invalid, either config or stream_info is a NULL pointer;<br>• Stream unsupported, maybe picture size exceeds or codec format is not supported in current version;<br>• Memory allocation failed when opening specific decoder;<br>• Failed when specific decoder initializes the FBM module; |

| | |
|---|---|
| | • Private data required by the specific decoder is not valid (see the Annex part D.3). |
| Special Requirements | Specific decoder may require some private data for initialization, see annex D.3 to understand the requirements of specific decoders for stream_info. |

Table-3 definition of structure vconfig_t

| typedef struct VLIB_CONFIG_INFO{ | | |
|---|---|---|
| u32 | max_video_width; | //* Maximum supported video picture width; |
| u32 | max_video_height; | //* Maximum supported video picture height; |
| u32 | max_output_width; | //* Maximum supported output picture width, if exceed, decoder force VE to scale down; |
| u32 | max_output_height; | //* Maximum supported output picture height, if exceed, decoder force VE to scale down; |
| u8 | scale_down_enable; | //* Whether use hardware scale down function to decode small pictures; |
| u8 | horizontal_scale_ratio; | //* Specifies the horizon scale ratio, 0: 1/1; 1: 1/2; 2: 1/4; 3: 1/8; |
| u8 | vertical_scale_ratio; | //* Specifies the vertical scale ratio, 0: 1/1; 1: 1/2; 2: 1/4; 3: 1/8; |
| u8 | rotate_enable; | //* Whether use the hardware rotate function to decode rotated pictures; |
| u8 | rotate_angle; | //* Clockwise rotate angle, 0: no ratate, 1: 90 degree; 2: 180; 3: 270; 4: horizon flip; 5: vertical flip; |
| u8 | use_maf; | //* Whether use MAF to help deinterlace; |
| u32 | max_memory_available; | //* The library should not allocate memory more than this size, see the recommended value //* for different format in Annex D.5, if max_memory_available == 0, the library will allocate as //* mush memory space as it needs. |
| }vconfig_t; | | |

Table-4 definition of structure vconfig_t

| typedef struct VIDEO_STREAM _INFO{ | | |
|---|---|---|
| stream_format_e | format; | //* codec such as H.264, MPEG-2, etc. stream_format_e is defined in file 'libve_typedef.h'. |
| stream_sub_format_e | sub_format; | //* codec sub-format, stream_sub_format_e is defined in file 'libve_typedef.h'. |
| container_format_e | container_format; | //* container format, such as AVI container_format_e is defined in file 'libve_typedef.h'. |
| u32 | video_width; | //* video picture width, if unknown please set it to 0. |
| u32 | video_height; | //* video picture height, if unknown please set it to 0. |
| u32 | frame_rate; | //* frame rate, multiplied by 1000, ex. 29.970 frames per second then frame_rate = 29970. |
| u32 | frame_duration; | //* how long a video picture should be show, in unit of micro-second. (us) |
| u32 | aspect_ratio; | //* pixel width to pixel height ratio, multiplied by 1000. |
| u32 | init_data_len; | //* data length of the private data for initializing a specific decoder. |
| u8* | init_data; | //* private initial data, a specific decoder may need it to startup. |
| }vstream_info_t; | | |

# libve_close

'libve_close' is method for the library users to close VE and release resource. User can tell whether to put the pictures kept by the specific decoder to FBM as valid or invalid frames. Before quiting to the caller, this method will release the FBM module through a method given by the IFBM interface.

In this method, the VE interrupt is disabled.

Table-5 ve_close

| | |
|---|---|
| Name | libve_close |
| Prototype | vresult_e libve_close (u8 flush_pictures, Handle ve); |
| Function | Close the library. |
| Return | • VRESULT_OK: success.<br>• VRESULT_ERR_LIBRARY_NOT_OPEN: the CSP is not opened yet.<br>vresult_e is defined in header file 'libve.h' |
| Input | u8 flush_pictures: put the pictures kept by specific decoder to FBM's display queue or as invalid pictures. |
| Input | Handle ve: handle of the VE device returned from the ve_open method. |
| Precondictions | 'libve_open' has been successfully called. |
| Success Guarantee | • All resource requested by the library is released;<br>• FBM modules are released;<br>• VE interrupt is disabled;<br>• To save power, VE clock is disabled. |
| Failed End Condition | 'libve_open' has not been successfully called yet. |
| Special Requirements | |

# libve_reset

'libve_reset' is a method for the library users to reset the VE. It will cause the specific decoder to clear its decoding status, flush pictures. After reset, all bitstream frames requested from the VBV will be discarded until a key bitstream frame is decoded.

Users can choose whether to put the frames kept by the specific decoder as valid frames for displaying, or as invalid frames returning to the FBM's empty buffer queue.

Table-6 libve_reset

| | |
|---|---|
| Name | libve_reset |
| Prototype | vresult_e libve_reset (u8 flush_pictures, Handle ve); |
| Function | Reset the library. |
| Return | • VRESULT_OK: success.<br>• VRESULT_ERR_LIBRARY_NOT_OPEN: the CSP is not opened yet.<br>vresult_e is defined in header file 'libve.h' |
| Input | u8 flush_pictures: put the pictures kept by specific decoder to FBM's display queue or as invalid pictures. |
| Input | Handle ve: handle of the VE device returned from the libve_open method. |
| Precondictions | 'libve_open' has been successfully called. |
| Success Guarantee | • Stream information is still kept by the specific decoder;<br>• Pictures kept by the specific decoder are all returned to the FBM;<br>• Bitstream frames kept by the library are returned to the VBV. |
| Failed End Condition | 'libve_open' has not been successfully called yet. |
| Special Requirements | |

## libve_set_vbv

Before decoding stream, the library user should call 'libve_set_vbv' to tell the VBV handle. The library use this information get bitstream frame and return bitstream frames, and calculate the bitstream offset of every decoded stream frame.

Table-7 libve_set_vbv

| | |
|---|---|
| Name | libve_set_vbv |
| Prototype | vresult_e libe_set_vbv (Handle vbv, Handle ve); |
| Function | Set VBV's bitstream buffer base address and buffer size to the library. |
| Return | • VRESULT_OK: success.<br>• VRESULT_ERR_LIBRARY_NOT_OPEN: the library is not opened yet.<br>vresult_e is defined in header file 'libve.h' |
| Input | Handle vbv: handle of the vbv module. |
| Input | Handle ve: handle of the VE device returned from the libve_open method. |
| Precondictions | 'libve_open' has been successfully called. |
| Success Guarantee | |
| Failed End Condition | 'libve_open' has not been successfully called yet. |
| Special Requirements | |

## libve_get_fbm

The library user needs a handle of the FBM instance when it request frames to display. As the FBM module is maintained by the specific decoder, the user should get the FBM handle from the library, using 'libve_get_fbm' method. If the FBM module is not initialized yet, 'libve_get_fbm' returns a NULL handle. If not only one FBM instance is initialized, the specific decoder should return the one which is storing pictures for display.

Table-8 libve_get_fbm

| | |
|---|---|
| Name | libve_get_fbm |
| Prototype | Handle libve_get_fbm (Handle ve); |
| Function | Get a handle of the FBM instance, in which pictures for display are stored. |
| Return | • Not NULL Handle: handle of the FBM instance.<br>• NULL Handle: FBM module is not initialized yet. |
| Input | Handle ve: handle of the VE device returned from the ve_open method. |
| Precondictions | FBM module has been initialized. |
| Success Guarantee | |
| Basic Flow | |
| Failed End Condition | FBM module has not been initialized yet. |
| Special Requirements | |

## libve_decode

'libve_decode' is a method used to decode bitstream. The library gets one bitstream frame from the VBV module

and decodes it. User can chooses to decode key frames only. If so, this method will discard stream data of P/B frames. User can also choose to discard overtime frames. In this case this method will compare the PTS of B frame and current time (passed in by user) to check whether to discard the stream data.

In this method, the specific decoder may request empty frame buffer from FBM module. If requests buffer failed, the library will return a fail code to tell there is no empty frame buffers.

If any frame decoded, the specific decoder should return a code to tell key frame or common frame decoded. If there is any frame can be displayed, the specific decoder should return or share it with the FBM module.

Table-9 libve_decode

| | |
|---|---|
| Name | libve_decode_stream |
| Prototype | vresult_e libve_decode (u8 keyframe_only, u8 skip_bframe, u32 cur_time, Handle ve); |
| Function | Decode one bitstream frame. |
| Return | VRESULT_OK: decode stream success but no frame decoded;<br>VRESULT_FRAME_DECODED: one common frame decoded;<br>VRESULT_KEYFRAME_DECODED: one key frame decoded;<br>VRESULT_ERR_FAIL: decode stream fail;<br>VRESULT_ERR_INVALID_STREAM: some error data in the stream, decode fail;<br>VRESULT_ERR_NO_MEMORY: allocate memory fail in this method;<br>VRESULT_ERR_NO_FRAMEBUFFER: request empty frame buffer fail in this method;<br>VRESULT_ERR_UNSUPPORTED: stream format is unsupported by this version of library;<br>VRESULT_ERR_LIBRARY_NOT_OPEN: 'libve_open' has not been successfully called yet. |
| Input | u8 keyframe_only: tell the library to decode key frame only; |
| Input | u8 skip_bframe: tell the library to skip B frame if it is overtime; |
| Input | u32 cur_time: current time, used to compare with PTS when decoding B frame; |
| Input | Handle ve: handle of the VE device returned from the libve_open method. |
| Precondictions | 'libve_open' has been successfully called;<br>The VBV module has video bitstream frames;<br>The FBM module has empty frame buffers; |
| Success Guarantee | • Any frame ready for display is return to or shared with FBM;<br>• If valid PTS is available, PTS should be set to each frame, if not available, PTS must be set to 0xffffffff;<br>• PCR value is passed from the bitstream frame to the output picture; |
| Failed End Condition | • Parameter invalid, either stream or ve is NULL;<br>• Stream unsupported, maybe picture size exceeds or codec profile is not supported in current version;<br>• Memory allocation failed;<br>• Failed to get empty frame buffer |
| Special Requirements | Each specific decoder has some requirement on the bitstream frame format, see the annex part D.4 to know what format should be followed for each decoder. |

## libve_get_stream_info

Sometimes the stream information like picture size is stored in stream data, and is not available to the library user until decoded by the specific decoder. In this case, the library user can call 'libve_get_stream_info' get the stream information.

Table-10 libve_get_stream_info

| | |
|---|---|
| Name | libve_get_stream_info |
| Prototype | vresult_e libve_get_stream_info (vstream_info_t* vstream_info, Handle ve); |
| Function | Get stream information from the CSP. |
| Return | • VRESULT_OK: success.<br>• VRESULT_ERR_INVALID_PARAM: either vstream_info or ve is NULL pointer;<br>• VRESULT_ERR_LIBRARY_NOT_OPEN: the CSP is not opened yet.<br>vresult_e is defined in header file 'libve.h' |
| Input | vstream_info_t* vstream_info: space to store video stream information. |
| Input | Handle ve: handle of the VE device returned from the ve_open method. |
| Precondictions | 'libve_open' has been successfully called. |
| Success Guarantee | The library store the informations in vstream_info, unknown things are set to zero; |
| Failed End Condition | • Parameter invalid, either vstream_info or ve is a NULL pointer;<br>• 've_open' has not been successfully called yet; |
| Special Requirements | |

# libve_get_version

'libve_get_version' returns a texture string describing the library's version information, like "Allwinner uni-v-vecoder version 1000". The digits "1000" describe software version.

Table-11 libve_get_version

| | |
|---|---|
| Name | libve_get_version |
| Prototype | u8* libve_get_version (Handle ve); |
| Function | Get version string of the VE library. |
| Return | A texture string in ASCII coding, describing the library's version information. |
| Input | Handle ve: handle of the VE device returned from the ve_open method. |
| Precondictions | |
| Success Guarantee | |
| Basic Flow | |
| Failed End Condition | Always success. |
| Special Requirements | |

# libve_get_last_error

If library operations return a fail result, the user can use 'libve_get_last_error' to get a texture string to know what is wrong.

Table-12 libve_get_last_error

| | |
|---|---|
| Name | libve_get_version |
| Prototype | u8* ve_get_last_error (Handle ve); |
| Function | Get error description. |
| Return | A texture string in ASCII coding, describing some information about the last error. |
| Input | Handle ve: handle of the VE device returned from the ve_open method. |

| | |
|---|---|
| Precondictions | |
| Success Guarantee | |
| Basic Flow | |
| Failed End Condition | Always success. |
| Special Requirements | |

## libve_flush

'libve_flush' is a method for the library users to flush decoded pictures to FBM. It will cause the specific decoder to flush pictures. After flush, all bitstream frames requested from the VBV will be discarded until a key bitstream frame is decoded.

Users can choose whether to put the frames kept by the specific decoder as valid frames for displaying, or as invalid frames returning to the FBM's empty buffer queue.

Table-13 libve_flush

| | |
|---|---|
| Name | libve_flush |
| Prototype | void libve_flush (u8 flush_pictures, Handle ve); |
| Function | Flush all decoded pictures to FBM module. |
| Return | No. |
| Input | u8 flush_pictures: put the pictures kept by specific decoder to FBM's display queue or as invalid pictures. |
| Input | Handle ve: handle of the VE device returned from the libve_open method. |
| Precondictions | • 'libve_open' has been successfully called; |
| Success Guarantee | |
| Basic Flow | |
| Failed End Condition | |
| Special Requirements | |

# C.2. Input Interface for Libve

As showed in Figure 1, the libve library is build upon an adapter level to make itself be application independent. In this adapter level, four interfacies have to be implemented to support the library. The first interface is named 'IVE'. It is designed for operations of Video Engine controlling issue. The second interface is named 'IFBM'. It is for the library to access frame buffers. The third interface is named 'IOS'. It is designed to be respondsible of OS based functions such as heap memory allocation. The last interface is 'IVBV'. It is designed to be respondsible of video bitstream frame accessing. These interfacies are defined in the header file 'libve_adapter.h'.

Below we discuss the prototype of these interfacies. The implementation of these interfacies is left to the library user.

## C2.1. The VE controlling interface – IVE

The interface IVE is a set of methods responsible for VE controlling, such as open or close pll to VE module, enable or disable VE interrupt, and reset VE hardware module. The prototype of IVE is defined as below.

In our Linux platform, the operations on VE controlling (such as opening pll for VE, enable the interrupt, getting VE's register base address) are implemented in the VE device driver (cedar_dev.ko, comprised in the Linux BSP). So when implementing the 'IVE' interface, you may need to use the VE device driver.

Table-14 definition of IVEControl_t

| typedef struct VE_CONTROL_INTERFACE{ | | |
|---|---|---|
| VE_RESET_HARDWARE | ve_reset_hardware; | //* method to reset VE hardware; |
| VE_ENABLE_CLOCK | ve_enable_clock; | //* method to enable or disable pll to VE hardware module; |
| VE_ENABLE_INTR | ve_enable_intr; | //* method to enable or disable VE interrupt; |
| VE_WAIT_INTR | ve_wait_intr; | //* method to wait VE interrupt coming; |
| VE_GET_REG_BASE_ADDR | ve_get_reg_base_addr; | //* method to get the base address of VE registers; |
| VE_GET_MEMTYPE | ve_get_memtype; | //* method to get DRAM type, such as DDR-2 16bits, DDR-1 32bits, etc; |
| }IVEControl_t; | | |

### ve_reset_hardware

The library use 've_reset_hardware' to reset the VE hardware module. In our Linux BSP, the VE's device driver (cedar_dev.ko) implements the reset operation. You can use the 'IOCTL_RESET_VE' IO command of the driver to implement 've_reset_hardware' method. Refer to the header file 'sun3i_cedar.h' to learn how to use the VE device driver.

Table-15 ve_reset_hardware

| Name | ve_reset_hardware |
|---|---|
| Prototype | void ve_reset_hardware (void); |
| Function | Reset the VE hardware module. |
| Return | No |
| Input | No |

## ve_enable_clock

The library use 've_enable_clock' to enable or disable clock to VE hardware module and set the clock speed. In our Linux BSP, the VE's device driver (cedar_dev.ko) implements the VE's enable/disable control operation. Look up the 'IOCTL_ENABLE_VE', 'IOCTL_SET_VE_FREQ' and 'IOCTL_DISABLE_VE' IO command in the header file 'sun3i_cedar.h'.

Table-16 ve_enable_clock

| Name | ve_enable_clock |
|---|---|
| Prototype | void ve_enable_clock (u8 enable , u32 speed); |
| Function | Enable or disable the VE clock and set clock speed. |
| Return | No |
| Input | u8 enable: '0' means disable VE clock, '1' means enable VE clock; |
| Input | u32 speed: clock speed in unit of Hz. |

## ve_enable_intr

The library use 've_enable_intr' to enable or disable VE interrupt. When VE interrupt is enabled, an interrupt service routine should be register to system to serve the interrupt. In this interrupt routine, you just clear the specific decoder's interrupt enable bits and return.

In our Linux BSP for F20, the interrupt serve routine has been implemented inside the VE's device driver (cedar_dev.ko). This ISR clear the VE's interrupt enable bits to disable the interrupt, and leave the VE's status bits to be handled by Libve. The VE interrupt is enabled when the device driver 'cedar_dev.ko' is installed. So you may implement nothing for this method.

Table-17 ve_enable_intr

| Name | ve_enable_intr |
|---|---|
| Prototype | void ve_enable_intr (u8 enable); |
| Function | Enable or disable the VE interrupt. |
| Return | No |
| Input | u8 enable: '0' means disable VE interrupt, '1' means enable VE interrupt; |

## ve_wait_intr

The library use 've_wait_intr' to wait for VE interrupt coming. This method should suspend the thread until a VE interrupt comes. In our Linux BSP, the VE's interrupt is handled by the device driver 'cedar_dev.ko'. You should use the IO command 'IOCTL_WAIT_VE' of this driver to wait for VE's interrupt. This IO command suspends the thread until a VE interrupt is handled.

Table-18 ve_wait_intr

| | |
|---|---|
| Name | ve_wait_intr |
| Prototype | s32 ve_wait_intr (void); |
| Function | Pend for VE interrupt coming. |
| Return | '0' means a VE interrupt comes, '-1' means wait time out but VE interrupt still not comes. |
| Input | No |

## ve_get_reg_base_addr

The library use 've_get_reg_base_addr' to get the virtual base address of the VE registers. In our Linux BSP, you should map the VE's register address space from the 'cedar_dev.ko' device driver. Like below:

```
fd = open("/dev/cedar_dev", O_RDWR);
ioctl(fd, IOCTL_GET_ENV_INFO, (unsigned long)& env_info);
env_info.address_macc = (unsigned int)mmap(NULL, 2048, PROT_READ | PROT_WRITE, MAP_SHARED, fd, env_info.address_macc);
ve_reg_base = env_info.address_macc;
```

Table-19 ve_get_reg_base_addr

| | |
|---|---|
| Name | ve_get_reg_base_addr |
| Prototype | u32 ve_get_reg_base_addr (void); |
| Function | Get the virtual base address of VE registers. |
| Return | Base address of VE registers. |
| Input | No |

## ve_get_memtype

The library use 've_get_memtype' to get the DRAM type. VE hardware needs to know this information to select the DRAM accessing method. The memory type is defined in header file 'libve_typedef.h'. Currently six types are defined. They are MEMTYPE_DDR1_16BITS, MEMTYPE_DDR1_32BITS, MEMTYPE_DDR2_16BITS, MEMTYPE_DDR2_32BITS, MEMTYPE_DDR3_16BITS and MEMTYPE_DDR3_32BITS.

Table-20 ve_get_memtype

| | |
|---|---|
| Name | ve_get_memtype |
| Prototype | memtype_e ve_get_memtype (void); |
| Function | Get the DRAM type, VE hardware module use this information to choose a DRAM accessing method. |
| Return | • MEMTYPE_DDR1_16BITS: DDR-1 DRAM with 16 bits bus width;<br>• MEMTYPE_DDR1_32BITS: DDR-1 DRAM with 32 bits bus width;<br>• MEMTYPE_DDR2_16BITS: DDR-2 DRAM with 16 bits bus width;<br>• MEMTYPE_DDR2_32BITS: DDR-2 DRAM with 32 bits bus width.<br>• MEMTYPE_DDR3_16BITS: DDR-3 DRAM with 16 bits bus width.<br>• MEMTYPE_DDR3_32BITS: DDR-3 DRAM with 32 bits bus width. |
| Input | No |

## C2.2. The Frame Buffer Manager interface – IFBM

The interface IFBM is a set of methods responsible for frame buffer accessing. The prototype of IFBM is defined as below:

Table-21 definition of IFBM_t

| | | |
|---|---|---|
| typedef struct FRAME_BUFFER_MANAGE_INTERFACE{ | | |
| FBM_INIT | fbm_init; | //* method to create a FBM instance; |
| FBM_RELEASE | fbm_release; | //* method to release a FBM instance; |
| FBM_REQUEST_FRAME | fbm_request_frame; | //* method to request an empty frame from a FBM instance; |
| FBM_RETURN_FRAME | fbm_return_frame; | //* method to return a frame to a FBM instance; |
| FBM_SHARE_FRAME | fbm_share_frame; | //* method to share a frame with a FBM instance; |
| }IFBM_t; | | |

### fbm_init

The specific decoder uses this method to initialize a FBM instance. If scale down or rotate function is opened, two FBM instance will be initialized, one for storing original picture and the other for storing scaled or rotated picture.

Table-22 fbm_init

| | |
|---|---|
| Name | fbm_init |
| Prototype | Handle fbm_init(u32 num_frames, u32 frame_width, u32 frame_height, pixel_format_e format); |
| Function | Initialize a FBM instance for storing pictures. |
| Return | • NULL: failed;<br>• Not NULL Handle: Handle of a FBM instance; |
| Input | num_frames: how many frame buffers to allocate; |
| Input | frame_width: the stored picture's width; |
| Input | frame_height: the stored picture's height; |
| Input | format: pixel format of the stored picture, such as ARGB or YUV420, etc; |

## fbm_release

The specific decoder uses this method to release FBM instance. When the decoder quit, this method will be called to release frame buffers.

Table-23 fbm_release

| | |
|---|---|
| Name | fbm_release |
| Prototype | void fbm_release(Handle fbm); |
| Function | Release an instance of FBM module. |
| Return | No |
| Input | fbm: the handle of a FBM instance that was returned by fbm_init() methed; |

## fbm_request_frame

The specific decoder uses this method to request an empty frame buffer for decoding.

Table-24 fbm_request_frame

| | |
|---|---|
| Name | fbm_request_frame |
| Prototype | vpicture_t* fbm_request_frame(Handle fbm); |
| Function | The specific decoder request an empty frame buffer from the FBM module. |
| Return | • NULL: failed to get an empty frame buffer;<br>• Not NULL pointer: Handle of an empty frame buffer; |
| Input | fbm: the handle of a FBM instance that was returned by fbm_init() methed; |

## fbm_return_frame

The specific decoder uses this method to return a frame that is no longer needed.

Table-25 fbm_return_frame

| | |
|---|---|
| Name | fbm_request_frame |
| Prototype | void fbm_return_frame(vpicture_t* frame, u8 valid, Handle fbm); |
| Function | The specific decoder returns a frame to the FBM module. |
| Return | No. |
| Input | frame: the frame to return; |
| Input | valid: whether the returned frame is valid for display; |
| Input | fbm: the handle of a FBM instance that was returned by fbm_init() methed; |

The vpicture_t structure is defined in header file 'libve_typedef.h' as below:

Table-26 definition of vpicture_t

| | | |
|---|---|---|
| typedef struct VIDEO_PICTURE{ | | |
| u32 | id; | //* index of the frame, set by the FBM module itself, other program should not change it; |

| | | |
|---|---|---|
| u32 | width; | //* the width of valid picture content, set by the specific decoder; |
| u32 | height; | //* the height of valid picture content, set by the specific decoder; |
| u8 | rotate_angle; | //* how this picture has been rotated, 0: no ratate, 1: 90 degree; 2: 180; 3: 270; 4: horizon<br>//* flip; 5: vertical flip; |
| u8 | horizontal_scale_ratio; | //* what ratio this picture has been scaled down at horizontal direction, 0: 1/1, 1: 1/2,<br>//* 2: 1/4, 3: 1/8; |
| u8 | vertical_scale_ratio; | //* what ratio this picture has been scaled down at vertical direction, 0: 1/1, 1: 1/2,<br>//* 2: 1/4, 3: 1/8; |
| u32 | store_width; | //* the stored picture's width in memory, may be a little bigger than 'width' as it have to be<br>//* a multiple of '32'; |
| u32 | store_height; | //* the stored picture's height in memory, may be a little bigger than 'height' as it have<br>//* to be a multiple of '32'; |
| u32 | frame_rate; | //* frame rate of the video stream, the video render module may need this field; |
| u32 | aspect_ratio; | //* aspect ration of this picture, it must be set by the specific decoder; |
| u8 | is_progressive; | //* whether this picture is interlace structure or progressive structure; |
| u8 | top_field_first; | //* if this picture is interlace structure, whether to display the top field first; |
| u8 | repeat_top_field; | //* if this picture is interlace structure, whether to repeat the top field when display; |
| u8 | repeat_bottom_field; | //* if this picture is interlace structure, whether to repeat the bottom field when display; |
| pixel_format_e | pixel_format; | //* the pixel format of this picture, such as ARGB or YUV420, etc; |
| u32 | pts; | //* the presentation time of this picture, must be set by the specific decoder. If unknown,<br>//* set it to be 0xffffffff; |
| u8* | y; | //* the content address of Y component in YUV format or R component in RGB format; |
| u8* | u; | //* the content address of U component in YUV format or G component in RGB format; |
| u8* | v; | //* the content address of V component in YUV format or B component in RGB format; |
| u8* | alpha; | //* the alpha data address when this picture is in ARGB format; |
| }vpicture_t; | | |

Before a picture is returned as a valid picture, the specific decoder has to set some field of the picture. These fields includes: width, height, aspect_ratio, is_progressive, top_field_first, repeat_top_field, repeat_bottom_field, pixel_format, pts, y, u, v and alpha.

## fbm_share_frame

The specific decoder uses this method to share a picture with the FBM module. When a picture is ready to display but the specific decoder still need it to be a reference picture for decoding, this picture should be shared between the specific decoder and the FBM module.

Table-27 fbm_share_frame

| Name | fbm_share_frame |
|---|---|
| Prototype | void fbm_return_frame(vpicture_t* frame, Handle fbm); |
| Function | The specific decoder shares a frame with the FBM module. |
| Return | No. |
| Input | frame: the frame to share; |

| Input | fbm: the handle of a FBM instance that was returned by fbm_init() methed; |
|---|---|

## C2.3. The OS interface – IOS

The interface IOS is a set of methods responsible for OS dependent functions that the library needs to use. The prototype of IOS is defined as below:

Table-31 definition of IOS_t

| typedef struct FRAME_BUFFER_MANAGE_INTERFACE{ | | |
|---|---|---|
| MEM_ALLOC | mem_alloc; | //* method to allocate buffer from heap; |
| MEM_FREE | mem_free; | //* method to release buffer to heap; |
| MEM_PALLOC | mem_palloc; | //* method to allocate physically continue buffer from heap; |
| MEM_PFREE | mem_pfree; | //* method to release buffer allocated by mem_palloc; |
| MEM_SET | mem_set; | //* method to set memory; |
| MEM_CPY | mem_cpy; | //* method to copy memory; |
| MEM_FLUSH_CACHE | mem_flush_cache; | //* method to flush cache content to memory; |
| MEM_GET_PHY_ADDR | mem_get_phy_addr; | //* method to change a virtual memory address to a get physical address; |
| SYS_PRINT | sys_print | //* method to printf string; |
| SYS_SLEEP | sys_sleep; | //* method to let a task itself sleep for some time; |
| }IOS_t; | | |

### mem_alloc

This method is used to allocate memory buffer, and it should work as the 'malloc' method defined in "ANCI C" standard.

Table-28 mem_alloc

| Name | mem_alloc |
|---|---|
| Prototype | void* mem_alloc(u32 size); |
| Function | Allocate memory buffer. |
| Return | Buffer address. |
| Input | size: buffer size in unit of byte; |

### mem_free

This method is used to free memory buffer that was allocated by 'mem_alloc'. It works as the 'free' method defined in "ANCI C" standard.

Table-29 mem_free

| Name | mem_free |
|---|---|
| Prototype | void mem_free(void* p); |
| Function | Free memory buffer. |

| | |
|---|---|
| Return | No. |
| Input | p: address of the buffer to free; |

## mem_palloc

In many systems, MMU may be used to remap scattered memory space as one continuous block of memory for CPU. This is called logic continue. For other hardware modules such as VE, memory space that is realy continuous is need. 'mem_palloc' is a method used to allocate physically continuous memory buffer.

Table-30 mem_palloc

| | |
|---|---|
| Name | mem_palloc |
| Prototype | void* mem_palloc(u32 size, u32 align); |
| Function | Allocate physically continuous memory buffer. |
| Return | Address of the buffer. |
| Input | size: buffer size in unit of bytes; |
| Input | align: requirement of the buffer's start address align, in unit of bytes; |

## mem_pfree

This method is used to free memory buffer that was allocated by 'mem_palloc'.

Table-31 mem_pfree

| | |
|---|---|
| Name | mem_pfree |
| Prototype | void mem_pfree(void* p); |
| Function | Free memory buffer allocated by 'mem_palloc'. |
| Return | No. |
| Input | p: address of the buffer to free; |

## mem_set

This method works as the 'memset' function defined in 'ANCI C' standard.

Table-32 mem_set

| | |
|---|---|
| Name | mem_set |
| Prototype | void mem_set(void* mem, u32 value, u32 size); |
| Function | Set every byte of a buffer to be 'value'. |
| Return | No. |
| Input | mem: address of the buffer to set; |
| Input | value: byte value to set; |

| | |
|---|---|
| Input | size: number of bytes in the buffer to set; |

## mem_cpy

This method works as the 'memcpy' function defined in 'ANCI C' standard.

Table-33 mem_cpy

| | |
|---|---|
| Name | mem_cpy |
| Prototype | void mem_cpy(void* dst, void* src, u32 size); |
| Function | Copy content from buffer 'src' to buffer 'dst'. |
| Return | No. |
| Input | dst: address of the destinate buffer; |
| Input | src: address of the content source; |
| Input | size: how many bytes to copy from 'src' to 'dst'; |

## mem_flush_cathe

This method is used to flush content from CPU's cathe to DRAM, so other hardware modules can use this content.

Table-34 mem_flush_cathe

| | |
|---|---|
| Name | mem_flush_cathe |
| Prototype | void mem_flush_cathe(void* mem, u32 size); |
| Function | Flush content from cathe to DRAM. |
| Return | No. |
| Input | mem: address of the cached memory; |
| Input | size: how many bytes to flush; |

## mem_get_phy_addr

This method is used to transform a virtual memory address to a physical address that hardware modules can use. A physical memory address may be mapped by the MMU to a virtual memory address, but hardware modules may access memory without using the MMU, so physical address is need.

Table-35 mem_get_phy_addr

| | |
|---|---|
| Name | mem_get_phy_addr |
| Prototype | u32 mem_get_phy_addr (u32 virtual_addr); |
| Function | Convert a virtual memory address to a physical address that hardware modules can use. |
| Return | The physical memory address. |
| Input | virtual_addr: the virtual memory address to convert; |

### sys_print

'sys_print' is a method that the library uses to print strings. Its implement is up to different application. Its prototype is defined as below:

Table-36 sys_print

| | |
|---|---|
| Name | sys_print |
| Prototype | s32 sys_print(u8* format, …); |
| Function | Printing string, act as the 'printf' function defined in 'ANCI C' standard. |
| Return | 0 if success and -1 if failed. |
| Input | format: the format string, such as "pinrt a number %d and a string %s"; |
| Input | …: other parameters depend on 'format'. |

### sys_sleep

'sys_sleep' is a method to make a thread sleep for some time.

Table-37 sys_sleep

| | |
|---|---|
| Name | sys_sleep |
| Prototype | Void sys_sleep(u32 ms); |
| Function | A thread calls this method to sleep for some time. |
| Return | No. |
| Input | ms: milliseconds sleep; |

## C2.4. The VBV interface – IVBV

The VBV is a module to manage bitstream frames. It maintains a loop buffer to receive bitstream data from a file parser, and share this loop buffer with VE to decode bitstream.

The IVBV interface is a set of methods for Libve to access video bitstream frames. The prototype of IVBV is defined below.

Table-38 definition of IVBV_t

| | | |
|---|---|---|
| typedef struct BITSTREAM_FRAME_MANAGE_INTERFACE{ | | |
| VBV_REQUEST_BITSTREAM_FRAME | vbv_request_bitstream_frame; | //* decode request one bitstream frame from VBV; |
| VBV_RETURN_BITSTREAM_FRAME | vbv_return_bitstream_frame; | //* decode didn't decode the stream yet, return it to the VBV; |
| VBV_FLUSH_BITSTREAM_FRAME | vbv_flush_bitstream_frame; | //* decoder has decode the stream, tell the VBV to flush it; |
| VBV_GET_BASE_ADDR | vbv_get_base_addr; | //* get the vbv loop buffer start address; |
| VBV_GET_SIZE | vbv_get_size; | //* get the vbv loop buffer size in unit of byte; |
| }IVBV_t; | | |

## vbv_request_bitstream_frame

The libve use this method to get one bitstream frame for decoding. This happens when libve_decode method is called. For some decoders like MPEG2, H264 and VC-1, libve may request more than one bitstream frames.

Table-39 vbv_request_bitstream_frame

| | |
|---|---|
| Name | vbv_request_bitstream_frame |
| Prototype | vstream_data_t* vbv_request_bitstream_frame(Handle vbv); |
| Function | Request one bitstream frame from the VBV module. |
| Return | Pointer to a Bitstream frame, NULL if fail. |
| Input | Handle vbv: handle of the vbv module, libve get this handle when 'libve_set_vbv' is called. |

## vbv_return_bitstream_frame

When one bitstream frame is requested by libve but not decoded yet, it will be return to the VBV module for next request. This may happens in the situation that one bitstream frame is requested by libve get no empty picture frame buffer for decoding. The VBV module should keep this bitstream frame again and pass it to the decoder at next request operation.

Table-40 vbv_return_bitstream_frame

| | |
|---|---|
| Name | vbv_return_bitstream_frame |
| Prototype | void* vbv_return_bitstream_frame(vstream_data_t* stream, Handle vbv); |
| Function | Return bitstream frame to the VBV module. |
| Return | No. |
| Input | vstream_data_t* stream: pointer to a Bitstream frame; |
| Input | Handle vbv: handle of the vbv module, libve get this handle when 'libve_set_vbv' is called. |

## vbv_flush_bitstream_frame

After one bitstream is decoded by libve, it should be flush frome the VBV module to release memory space for the new coming bitstream frames. Libve use this method to tell the VBV module that one bitstream frame is finished and should be flush out.

Table-41 vbv_flush_bitstream_frame

| | |
|---|---|
| Name | vbv_flush_bitstream_frame |
| Prototype | void* vbv_flush_bitstream_frame(vstream_data_t* stream, Handle vbv); |
| Function | Flush bitstream frame out from the VBV module. |
| Return | No. |
| Input | vstream_data_t* stream: pointer to a Bitstream frame; |

| | |
|---|---|
| Input | Handle vbv: handle of the vbv module, libve get this handle when 'libve_set_vbv' is called. |

## vbv_get_base_addr

As libve shares a loop buffer with the VBV module for bitstream data, it needs to know the buffer's start address and the buffer's size. That because one bitstream frame data may cross the loop buffer's end and loop back to the start.

Table-42 vbv_get_base_addr

| | |
|---|---|
| Name | vbv_get_base_addr |
| Prototype | u8* vbv_get_base_addr(Handle vbv); |
| Function | Get the VBV module's loop buffer base address. |
| Return | The VBV module's loop buffer base address; |
| Input | Handle vbv: handle of the vbv module, libve get this handle when 'libve_set_vbv' is called. |

## vbv_get_size

As libve shares a loop buffer with the VBV module for bitstream data, it needs to know the buffer's start address and the buffer's size. That because one bitstream frame data may cross the loop buffer's end and loop back to the start.

Table-43 vbv_get_size

| | |
|---|---|
| Name | vbv_get_size |
| Prototype | u32 vbv_get_size (Handle vbv); |
| Function | Get the VBV module's loop buffer size |
| Return | The VBV module's loop buffer size; |
| Input | Handle vbv: handle of the vbv module, libve get this handle when 'libve_set_vbv' is called. |

# Annex

## D.1. A recommended design of the VBV module

The VBV module is designed to mange video bitstream frame data stored in a loop buffer. There are mainly two threads operating on the loop buffer, the data pasing thread and the decode thread. The data parsing thread write video stream frames into the loop buffer. It request buffer from the VBV module, write video stream data into the buffer and then tell the VBV module that it has update the buffer with video data. The decoding thread decodes video bitstream data and output pictures. It request bitstream frame from the VBV module, decode the data and then tell the VBV module to flush the bitstream frame to release buffer.

In our VBV module, the data parsing thread uses the 'vbv_request_buffer' method to request buffer from the VBV module. When one bitstream frame is ready, the data parsing thread uses the 'vbv_add_stream' method to add one frame of bitstream to the VBV module. The decoding thread uses the 'vbv_request_stream_frame' method to request bitstream frame for decoding. If one bitstream frame is decoded, the decoding thread calls the 'vbv_flush_bitstream_frame' method to flush it to release buffer. If one bitstream frame is not decoded, the decoding thread may call the 'vbv_return_bitstream_frame' method to return it to the VBV module.

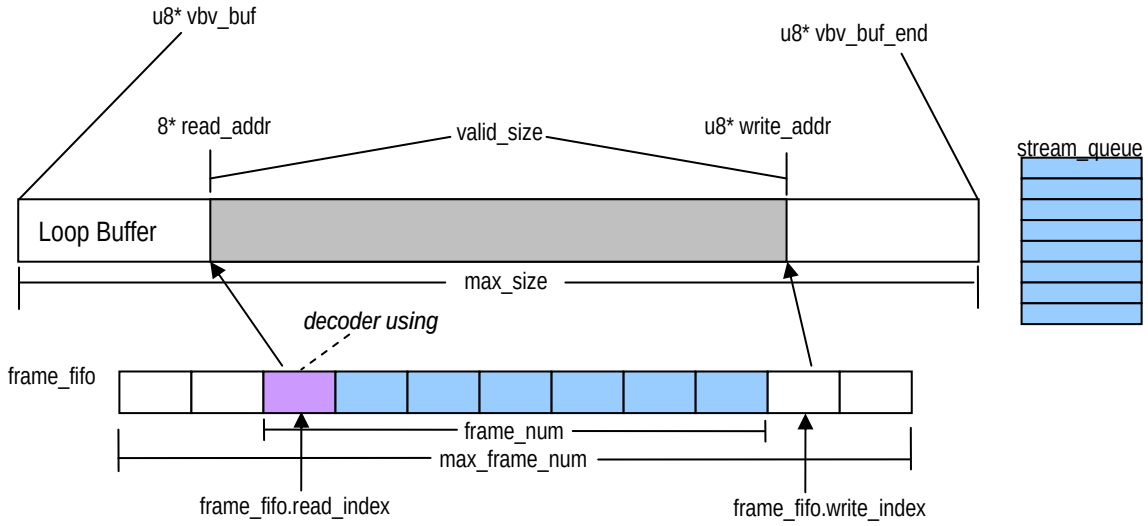


Figure 8 Architecture of VBV module

The architecture of VBV module is described in the figure above. For details about the implementation, please refer to the source code 'vbv.h' and 'vbv.c'.

# D.2. A recommended design of the FBM module

The FBM module is a module for picture frame buffer managing. There are mainly two threads operating on the FBM module, the decoding thread and the video render thread. The decoding thread requests empty frame buffer from the FBM module, decodes picture data to the buffer. If one picture frame can be send to displayed, the decoder thread will return or share (in case that the decoder still need the frame as a reference frame) the frame buffer to the FBM module. The render thread requests pictures from the FBM module, displays the picture and then returns it to the FBM module.

In our FBM module, the decoding thread uses 'fbm_decoder_request_frame' method to request empty frame buffer. When one picture is decoded and the decoder does not need this picture anymore, the decoder calls 'fbm_decoder_return_frame' to return the picture to the FBM module. If the decoded picture should be display but the decoder still need the picture as a reference frame, the 'fbm_decoder_share_frame' method will be called by the decoder to share this frame.

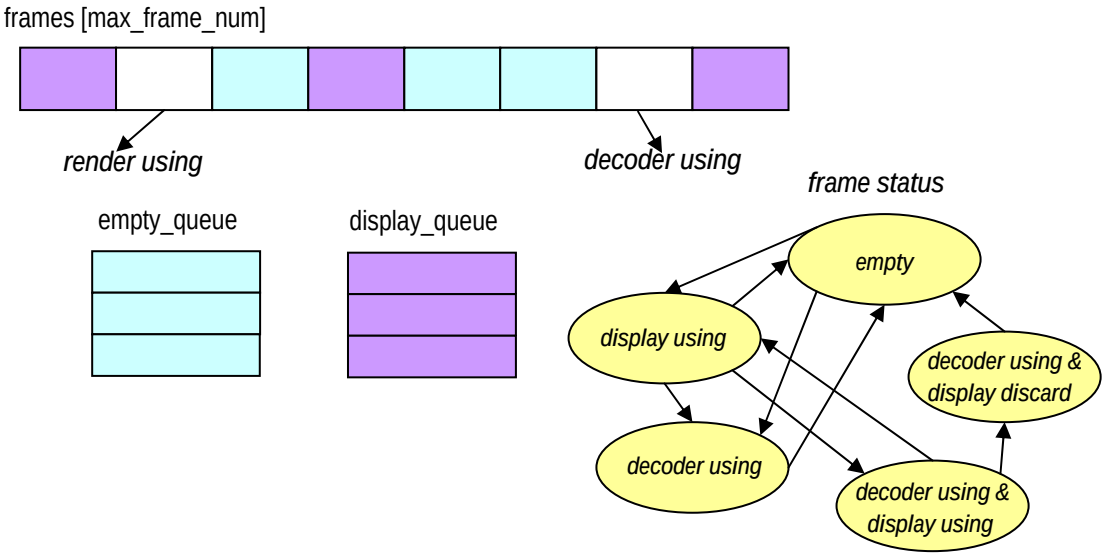


Figure 9 Architecture of FBM module

The architecture of FBM module is described in the figure above. For details of implementation, please refer to the source code 'fbm.h' and 'fbm.c'.

# D.3. Requirement for initializing stream information

When opening libve, the stream information should be passed in to 'libve_open' method. The structure of stream information is defined as below:

Table-44 definition of structure vconfig_t

| typedef struct VIDEO_STREAM _INFO{ | | |
|---|---|---|
| stream_format_e | format; | //* codec such as H.264, MPEG-2, etc. stream_format_e is defined in file 'libve_typedef.h'. |
| stream_sub_format_e | sub_format; | //* codec sub-format, stream_sub_format_e is defined in file 'libve_typedef.h'. |
| container_format_e | container_format; | //* container format, such as AVI container_format_e is defined in file 'libve_typedef.h'. |
| u32 | video_width; | //* video picture width, if unknown please set it to 0. |
| u32 | video_height; | //* video picture height, if unknown please set it to 0. |
| u32 | frame_rate; | //* frame rate, multiplied by 1000, ex. 29.970 frames per second then frame_rate = 29970. |
| u32 | frame_duration; | //* how long a video picture should be show, in unit of micro-second. (us) |
| u32 | aspect_ratio; | //* pixel width to pixel height ratio, multiplied by 1000. |
| u32 | init_data_len; | //* data length of the private data for initializing a specific decoder. |
| u8* | init_data; | //* private initial data, a specific decoder may need it to startup. |
| }vstream_info_t; | | |

The 'format' field is used to decide which specific decoder in VE to use. It should be set with one enumeration value of stream_format_e defined in 'libve_typedef.h', such as *STREAM_FORMAT_MPEG2*.

The 'sub_format' field is important for MPEG4 decoder. In MPEG4 decoder, we implement several standards such as XVID, DIVX3, DIVX4, DIVX5, H263, VP6 and so on. This field should be set with one enumeration value of stream_sub_format_e.

The 'container_format' is used to identify in which file format is the video stream packeted. Just set it to one enumeration value of container_format_e defined in 'libve_typedef.h' according to the file type.

The 'video_width' and 'video_height' field are the source video picture size. For some file format, these fields may not be known at the beginning. In this case, set them to zero and let the decoder to find this information.

The 'frame_rate' field means how many pictures should be displayed in one second. This field will be passed through to the video render with every picture frame. Sometimes this field is also used to guess video picture presentation time when PTS is invalid.

The 'aspect_ratio' field means the ratio of the video source's pixel width and pixel height. If this field can not be parsed from the file, just set it to 1000. The decoder will handle this information and passes it through to the video render.

The 'init_data' field contains some private data for some decoder. Below we introduce what the init_data is for each decoder.

1. MPEG4:
   //* TODO.

2. H264:
   //* TODO.

3. MPEG1/2:
   //* TODO.

4. VC-1:
   //* TODO.

5. AVS:
   //* TODO.

6. MJPEG:
   //* TODO.

7. VP8:
   //* TODO.

## D.4. Requirement on bitstream format

## D.5. Recommendation of memory size for decoders