HDAL Programmer's Guide

<Revison history>

VERSION	Date	Contents	Created by	Approved by
0.0.1	2018-08-13	• Document Template, Table of contents created • Creation of the concept of HDAL (Overview) • Creation of HDAL Structure	Alex Sun	
		• Creation of each chapter-wise concents and addition of examples		

<Concept>
It describes the structure of HDAL (Hardware Device Abstraction Layer) and how to use it.

<Information>
Creator : iVot SW Platform
Status : Public
Audience : Project Leader, Developer, Customer
Pages : 69 Pages

Table of Contents

List of Figures

List of Tables

1. HDAL Overview

1.1. Definition and Purpose

HDAL is the acronym of Hardware Device Abstraction Layer. It replaces the part being occupied by HAL (Hardware Abstraction Layer) in existing SW Architecture for the device transparent from independent devices and platforms, includling the product, IP Camera, DVR and NVR.

For the existing HAL, it is created in the Chip-dependent structure, therefore when the chip is changed to Novatek or others, the APIs of HAL have to be changed depending on the change of chip at times. It is the object of HDAL to replace HAL and perform the function using Chip-independent Common API. Moreover, by providing only the header of HDAL API to chip venders and letting them implement the contents by themselves, the Middleware developers can implement this even though they are not aware of technological details. That is the HDAL is the common modules with common reused APIs, there is vendor interfaces for the customization for the dependent appliance, like AI or image quality modules.

Fig 1 is an example use case for live view and video record flows

The purpose of this specification is to explain the features of HDAL fully by describing the structure and the manual of HDAL.

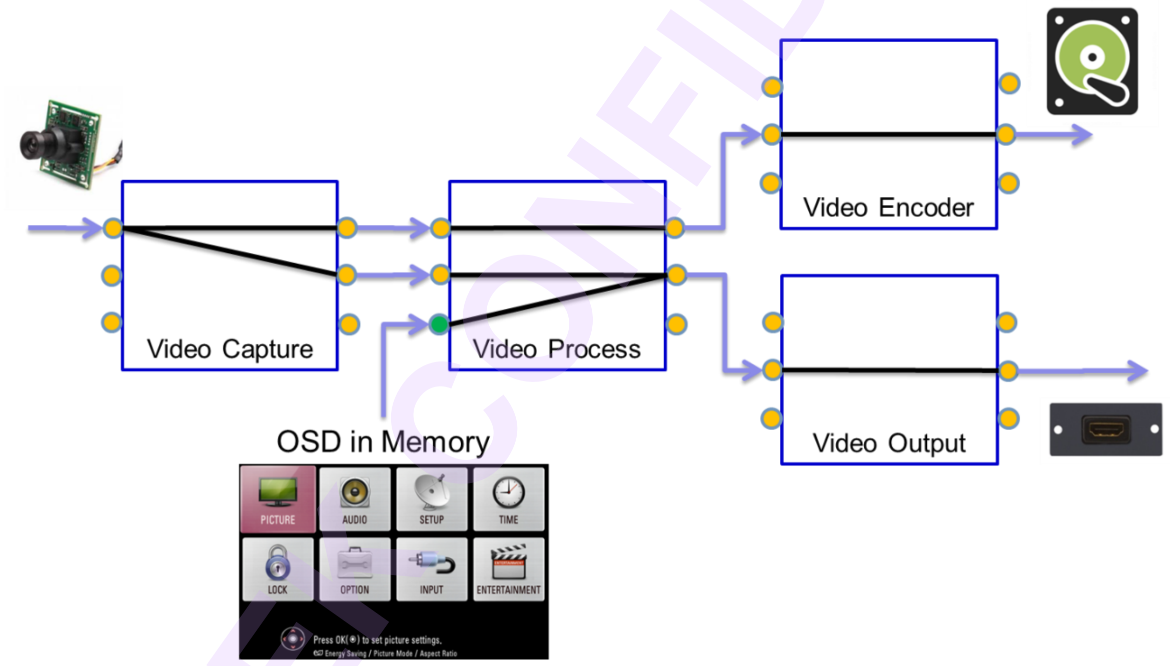

Fig 1 An example use case for live view and video record flows

1.2. Scope

This specification explains the structure of HDAL and describes the way of using HDAL with examples. However, this is not to explain how HDAL will be used for every scenario, but instead, the API of HDAL will be described.

1.3. Definition & Acronyms

- HDAL: Hardware Device Abstraction Layer
- S/W: Software
- H/W: Hardware
- M/W:Middleware
- HAL: Hardware Abstraction Layer

- D/D: Device Driver
- hd_: the prefix of HDAL module or API naming.

1.4. Responsibilities

Module Name	DVR Owner	IPC Owner	Approved
Project Leader	Foster	Jeah	
hd_audiocapture	Schumy	HM	
hd_audioenc	Schumy	Adam	
hd_audiodec	Schumy	Adam	
hd_audioout	Schumy	HM	
hd_videocapture	Foster	Ben	
hd_videoprocess	KL	Jeah	
hd_videoenc	Schumy	Boyan	
hd_videodec	KL	Boyan	
hd_videoout	Jerry	Janice	
hd_gfx	Jerry	YongChang	
hd_utl	Foster	HM	
hd_logger	KL	Niven	
hd_common	Harry	Harry	
hd_degug	KL	Niven	

1.5. Constraints

The development OS is the Linux. The definition for slim OS is coming soon.

1.6. References

- HDAL Design Proposal

2. HDAL Structure

2.1. HDAL S/W Architecture

HDAL is a layer located between TD and Chip Driver. HDAL is composed of 14 instances in total, and it is engineered to ease function realization at TD as it is abstracted by the function and not followed H/W structure. Fig 2 below shows the location of HDAL in SW Architecture, including Vendor and HDAL instances.

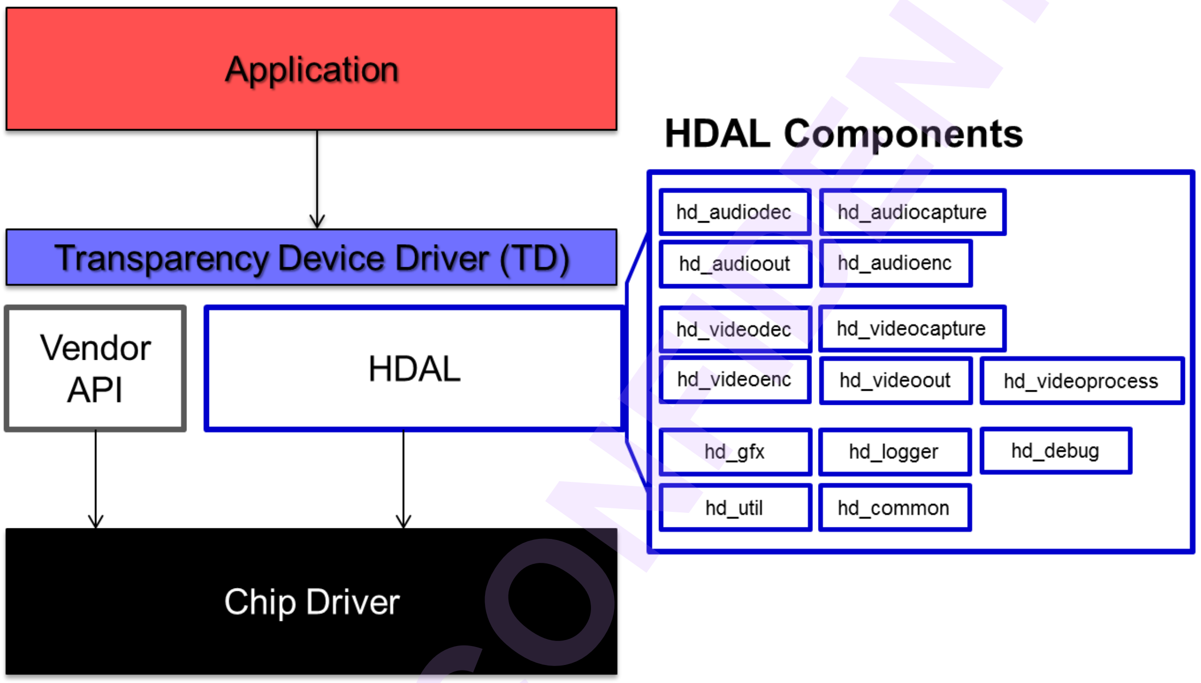

Fig 2 Location of HDAL in whole SW Architecture

Each HDAL instance realization range changes by the function of Chip which is the target of HDAL. Thus, Chip with Camera or Display functions missing might not employ HDAL instances on relevant functions. HDAL instance is HW Platform independent and provides common APIs and also is distinguished by function considering interface between the AV signal flow and the parent module.

2.2. HDAL Instance

2.2.1. HDAL Instance's basic structure

Each HDAL instance with streaming features has the following API at its basic.

Table 1 Basic API list

Function name	Description
hd_xxx_init()	Responsible for initializing the class, especially for the global setting. Generally it is Called only once at system initialization, or when memory state changes.
hd_xxx_uninit()	Responsible for uninitializing the global setting of the class. Generally it is Called only once at system uninitialization, or when memory state changes.
hd_xxx_open()	Responsible for opening an instance from the class. Generally it is Called after the class was initialized. The function is used to call for opening multiple instances.
hd_xxx_close()	Responsible for closing an instance of the class. Generally it is Called to close the instancesetting. The instance will be removed after calling this function.
hd_xxx_bind()	Each Instance is called up when it is connected with fore-end or back-end of Instance. Generallly, relevant Instance ID and front terminal Instance ID are used for input parameter, but for some instances, additional input parameter is needed. Call for hd_xxx_bind() function is only valid under Disconnect status and if tried to Connect to change connection status without disconnecting at Bind status, Error will occur.
hd_xxx_unbind()	It is called up when unbind is required after each instance is connected. Since only the concerned instance's ID will be given as parameter, the hd_xxx_unbind() disconnects the front and back connection as referring to the concerned Instance's ID.
hd_xxx_get()	It is used upon taking the information of each Instance's settings. This API is used when taking the previous setting information before calling up the hd_xxx_set() and then changing only the settings value to be changed. The using method will be displayed along with the hd_xxx_set() example.
hd_xxx_set()	It is used upon setting the information for each instance settings. There is the structure for the instance-wise setting information, in order to change the setting information, call the hd_xxx_get() to get the structure for the previous setting information and then change only the value which is wished to be changed and then call up the hd_xxx_set(). If the hd_xxx_get() is called up after calling up hd_xxx_init(), the settings value will be all entered in to the HD_INVALID. If the setting variable in the implementation part of the hd_xxx_set() is HD_INVALID, it should be implemented not to set the variable but to skip it.

The features which would be called up frequency by subsystem other than the basic APIs are definded as the Action API and exist for each Instance so that user can see the change as it would be executed immediately when being called up.

For taking example for hd_audioout, the following APIs exist. The Parameters for each API have been omitted on the convenience basis.

```
HD_RESULT hd_audioout_init();
HD_RESULT hd_audioout_open();
HD_RESULT hd_audioout_start();
HD_RESULT hd_audioout_stop();
HD_RESULT hd_audioout_close();

HD_RESULT hd_audioout_set();
HD_RESULT hd_audioout_get();

HD_RESULT hd_audioout_new_in_buf();
HD_RESULT hd_audioout_push_in_buf();
HD_RESULT hd_audioout_release_in_buf();
```

Among the APIs of above hd_audioout, the hd_audioout_new_in_buf(), hd_audioout_push_in_buf() and hd_audioout_release_in_buf(), are fallen under the Action API which is described above.

2.2.2. HDAL Instance description

The each HDAL Instance-wise features are as following. In Table 2, the columns with background light red are audio related modules, the columns with light green are video related modules and the columns without background color are system utility modules, like debug, graphics and logging functions.

Table 2 HDAL Instance-wise features

Instance name	Function
hd_audiocapture	As it is for the audio input features, it performs the feature of informing the audio input setting and state information.
hd_audioout	It performs the auxiriary output features of Audio. The Audio auxiliary output refers like to Analog Audio output and HDMI digital output.
hd_audioout	It performs the feature of processing the Audio signal and external input to be outputted from Audio Decoder which comes in like from I2S and the feature of controlling the Speaker or HDMI output.
hd_audiodec	It takes the charge of Audio Decoding and the feature of controlling Audio Decoder flow and format.
hd_audioenc	It provides the APIs which can control the Audio Encoder.
hd_videocapture	As it is for the video input features, it performs the feature of informing the video input state information.
hd_videoout	It performs the controlling related to Video output which is connected like to the Panel or HDMI and the Video Post controlling features
hd_videodec	It takes the charge of filtering the Video, Video Decoding and the feature of controlling video Decoder.
hd_videoenc	It provides the APIs which can control the Video Encoder.
hd_videoprocess	Being divided as the video processing feature separately, it is the API which sets the Resolution, Video Size and Video Position and performs the frame rate convertion feature. The main feature is video process controlling however, the features of which the video out are controlled separately should be performed by this instance.
hd_common	In charge of functions such as I2C/GPIO and general formatting including target configuration.
hd_debug	In charge of debug function, like HDAL Debug Menu entry point
hd_gfx	In charge of functions related to Graphic plane.
hd_log	Directive printf function to difference interfaces
hd_util	Supplies utility function which could be used in HDAL such as key input, Print etc.

2.3. Naming Convention

The general rule for the function name, macro name, enum name, structure name, variable name and variable type name which are used in HDAL is described.
Generally speaking, filenames follow the snake_case convention of the module they define. For example, MyApp should be defined inside the my_app.ex file. However, this is only a convention. At the end of the day, any filename can be used as they do not affect the compiled code in any way.

2.3.1. function

- API function

hd_<Module name>_<Part name><Get/Set/Verb><Noun words and phrases>

```
HD_RESULT hd_videoproc_unbind(HD_DAL self_id, HD_IO out_id);
```

- API private function (To be used internally only, not being used above)

hd_<Module name>_<Part name><Get/Set/Verb><Noun words and phrases>_p

```
HD_RESULT hd_videoproc_unbind_p(HD_DAL self_id, HD_IO out_id);
```

- Class member function

<Part name><Get/Set/Verb><Noun words and phrases>

```
HD_RESULT unbind(HD_DAL self_id, HD_IO out_id);
```

2.3.2. Macro

<Part name 0>_<Parnt name 1>_...<Noun>

2.3.3. Enumerated constant

```
typedef enum
  {
    HD_<Part name 0>_<Part name 1>_...<Noun>
    or
    HD_<Module name>_<Part name 0>_...<Noun>
  } Sd<Module name>_<Part name 0><Part name 1>...<Type/Mode/Status/Noun>_k
```

```
typedef struct _HD_VIDEOPROCESS_CROP {
        HD_CROP_MODE mode;
        HD_VIDEO_CROP win;
} HD_VIDEOPROCESS_CROP;

typedef enum _HD_VIDEOPROCESS_COLOR_SPACE {
        HD_VIDEOPROCESS_COLOR_FULL        = 0,    ///< full range
        HD_VIDEOPROCESS_COLOR_BT601       = 1,    ///< BT.601 (SDTV)
        HD_VIDEOPROCESS_COLOR_BT709       = 2,    ///< BT.709 (HDTV)
        HD_VIDEOPROCESS_COLOR_MAX,
        ENUM_DUMMY4WORD(HD_VIDEOPROCESS_COLOR_SPACE)
} HD_VIDEOPROCESS_COLOR_SPACE;
```

2.3.4. Structure

```
    typedef struct
  {
    ...
  } hd_<Module name>_<Part name 0><Part name 1>...<Info/header/Noun>_t
```

2.3.5. Variable

- The return variable is set to the HD_RESULT.
- it starts from small letter and the first character of the word starts as the capical letter.
- The member variable of Class start as m_
- Global variable : starts as g_
- Handle starts as h
- *(pointer) starts as p
- bool starts as b
- float starts as f (flag variable does not start as f, the word of flag should be specified in the name)
- enum starts as e
- array starts as a
- struct starts as s
- The Variable Naming Rule excluding the Class member variable and Global Variable will not be applied as being overlapped, but only one with high priority will be applied. The highest Priority is "Return variable dis set as SdResult." And the lowest priority is "Struct starts as s".
- The class member variable and Global variable are appled with another variable Naming Rule overlapped. E.g.) m_bEnable (For the bool as the member variable of Class)

2.3.6. Variable type

- The re-definition of Variable Type which is defined in C or C++.
- To be set as hd_<Redefinition Type>_t.
- The first character of redefinition Type is the capital letter.
- E.g) hd_videoprocess_settings_t, hd_videoout_videoformat_t

2.3.7. Standard abbreviation

Use the abbreviation as referring to the below table.

Table 3 Standard abbreviations

Original	After	Original	After
Version	Ver	Calculator	Calc
Specification	Spec	Manager	Mgr
String	Str	Application	App
Document	Doc	Synchronization	Sync
Command	Cmd	Memory	Mem
Device	Dev	Function	Func
Sequence	Seq	Enumeration	Enum
Source	Src	Constant	Const
Picture	Pic	System	Sys
On Screen Display	OSD	Configure	Config
Elec. Prog. Guide	EPG	Option	Opt
Character	Char	Set top box	STB
Standard	Std	Message	Msg

Audio/Video	AV	User Interface	UI
Multiplexer	MUX	Error	Err
Directory	Dir	Variable	Var
Transport Stream	TS	Environment	Env
Power	Pwr	Channel	Ch
Audio	Aud	Video	Vid

3. Instance of Streaming Module

3.1. Overview

Therea are 2 types of modules, the streaming and utility modules. The streaming modules are designed with are defined in the next section. If the modules are not related with streaming, we called utility modules. That is to perform, like initionalize the HDAL, graphics functions, debug functions and etc.
In this section, there describe the detail about the streaming modules, the audio and video. The operating flow are described in the following.
With streaming media, like IP Camera or Digital Video Recoder, a user does not have to wait for a complete file to play it. Because the media is sent in a continuous stream of data it can play as it arrives.Streaming module is the code to control the flow of video or audio content sent in compressed or raw form over the Internet or device. Here is the stream modules defined for audio input, video input, audio output, video output, audio encoder, video encoder, audio decoder, video decoder and video processing. There are described as the following about the functional.

- hd_audiocapture/ hd_audioout: Audio input and output related.
- hd_audioenc/hd_audiodec: Audio encode and decode related.
- hd_videocapture/ hd_videoout: Video input and output related.
- hd_videoenc/hd_videodec: Video encode and decode related.
- hd_videoprocess: Video process related.

3.2. HDAL API typical calling flow

In HDAL, there standardlized the API calling flow for typical streaming module. Developers can refer the calling flow, especially on setting the attribute of the module.
Fig 3 below shows the typical command colling flow for streaming module

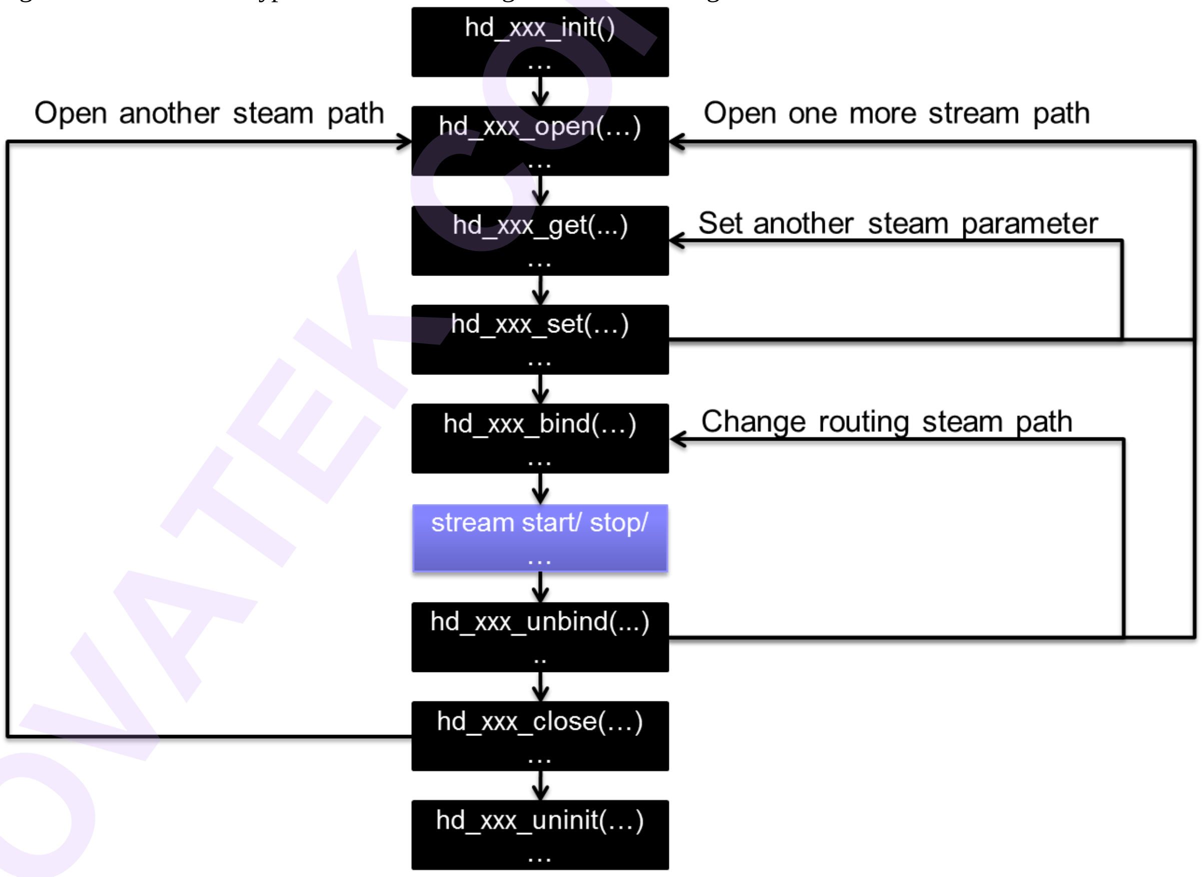

Fig 3 Typical calling flow for streaming module

3.3. Configuration inside the instance of streaming module

3.3.1. Opening the streaming module

A stream module is an abstract interface for working with streaming data in HDAL. The stream module provides a base API that makes it easy to build objects that implement the stream interface. While it is important to understand how streams work, the stream module itself is most useful for developers that are controlling the setting of stream instances. Developers who are primarily consuming stream instances will need to set the attributes of the stream module directly.
To enable/open the streaming instance, there are 3 parameter expose to external for data linking. When developers want to enable an input port to output port, we called it's a path. Developers shall call the hd_xxx_open() function (here xxx defined as stream module). There are 3 parameters for open streaming module, in port, out port and path ID. The path ID is the return value of calling hd_xxx_open() function, for example `hd_videocap_open`.
Fig 4 below shows an example of opening Video Capture. In this case, there open 2 paths with same source but to 2 different output ports. The mapping of the paths can be one in put port to many output ports, typical for video capture device. Or developers can map path as many input ports to one output port, like video out with picture in picture (PIP) case.

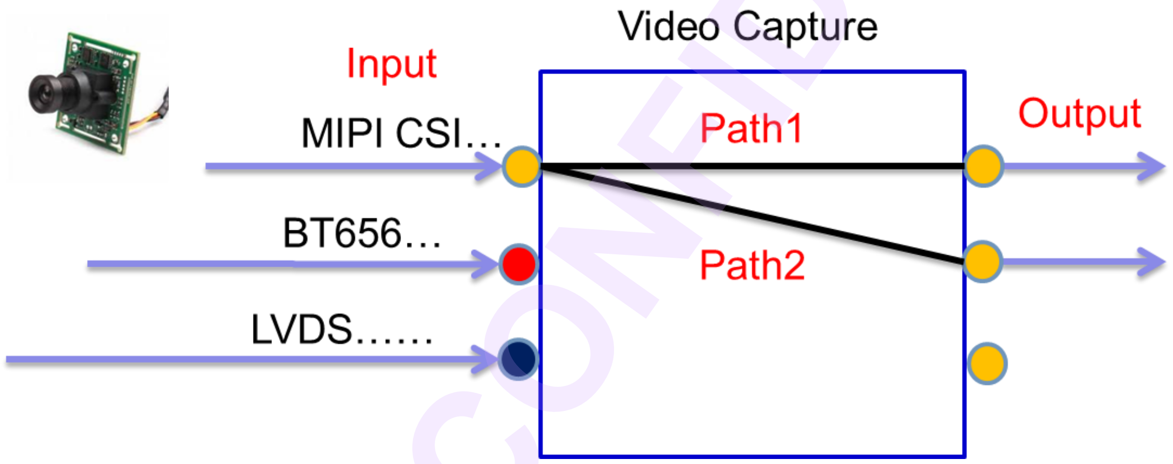

Fig 4 An example of opening Video Capture

```
    HD_RESULT ret;
     if((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_0, 
&cap0_path_id)) != HD_OK)
        return ret;
     if((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_1, 
&cap1_path_id)) != HD_OK)
        return ret;
    if((ret = hd_videoproc_open(HD_VIDEOPROC_0_IN_0, HD_VIDEOPROC_0_OUT_0, 
&vpe0_path_id)) != HD_OK)
        return ret;
    if((ret = hd_videoenc_open(HD_VIDEOENC_0_IN_0, HD_VIDEOENC_0_OUT_0, 
&enc0_path_id)) != HD_OK)
        return ret;
    if((ret = hd_videoenc_open(HD_VIDEOENC_0_IN_1, HD_VIDEOENC_0_OUT_1, 
&enc1_path_id)) != HD_OK)
        return ret;
    return HD_OK;
```

3.3.2. Setting the attribute of the streaming module

A configuration of stream instance consists of many parts, like frame rate, resolution, input/ output color type. Each instance in the configuration is represented by such an object, like video capture, video output, video processing. An attribute is a named property that can be get or set using the HDAL streaming API. The value of

an attribute can be an integer, a floating-point number, a string, an point reference, a boolean value, a list of values, or a mapping from values to other values.
The function of geting setting returns a handle and the point value of a specified attribute on the attribute. The below shows about get a path (cap0_path_id) attribute about HD_VIDEOCAP_PARAM_SYSCAPS. It get the width and hight from the video capture path.

```
    //init dim for main/sub streams
     hd_videocap_get(cap0_path_id, HD_VIDEOCAP_PARAM_SYSCAPS, &cap_syscaps);

    main_dim.w  = cap_syscaps.max_dim.w;
    main_dim.h  = cap_syscaps.max_dim.h;
    sub_dim.w  = cap_syscaps.max_dim.w / 2;
    sub_dim.h  = cap_syscaps.max_dim.h / 2;
```

The below shows about set a path (cap0_path_id) attribute about HD_VIDEOCAP_PARAM_SYSCAPS. The setting method modifys the specified attribute to an instance, and gives it the specified value.It set the attributes of the video capture path attribute in the example.

```
     HD_VIDEOENC_PATH_CONFIG enc_config;
     HD_VIDEOCAP_PATH_CONFIG cap_config;
     HD_VIDEOENC_IN video_in_param;
     HD_VIDEOENC_OUT enc_param;
     HD_H26XENC_RATE_CONTROL rc_param;
     HD_RESULT ret;

     //set main videocap out pool
     memset(&cap_config, 0x0, sizeof(HD_VIDEOCAP_PATH_CONFIG));
     cap_config.data_pool[0].mode = HD_VIDEOCAP_POOL_ENABLE;
     cap_config.data_pool[0].ddr_id = 0;
     cap_config.data_pool[0].pxlfmt = HD_VIDEO_PXLFMT_YUV420_NVX3;
     cap_config.data_pool[0].dim.w = main_dim.w;
     cap_config.data_pool[0].dim.h = main_dim.h;
     cap_config.data_pool[0].counts = HD_VIDEOCAP_SET_COUNT(4, 0);
     cap_config.data_pool[0].max_counts = HD_VIDEOCAP_SET_COUNT(4, 0);


     cap_config.data_pool[1].mode = HD_VIDEOCAP_POOL_DISABLE;
     cap_config.data_pool[2].mode = HD_VIDEOCAP_POOL_DISABLE;
     cap_config.data_pool[3].mode = HD_VIDEOCAP_POOL_DISABLE;

     ret = hd_videocap_set(cap0_path_id, HD_VIDEOCAP_PARAM_PATH_CONFIG, &cap_config);
     if (ret != HD_OK) {
          printf("hd_videocap_set for main pool fail\n");
          goto exit;
     }
```

3.4. Configuration between the streaming modules

The HDAL is the interfaces to run the streaming that is involved with direct rendering, like the behaviors of Direct Memory Acess. After initializing input port, output port and its pathes with setting its own resources, it prepares for the device interfaces related with the dedicated harware and waits for binding connection. Then, when a streaming rendering command binds by the output and input ports, the connection protocol will be established between the 2 binded streaming instances, and instance resources are allocated. This section describes the operations necessary to bring the module to a connected state with bind command.
A video output pipeline using only V4L2 (no DRM) is not supported at this time.

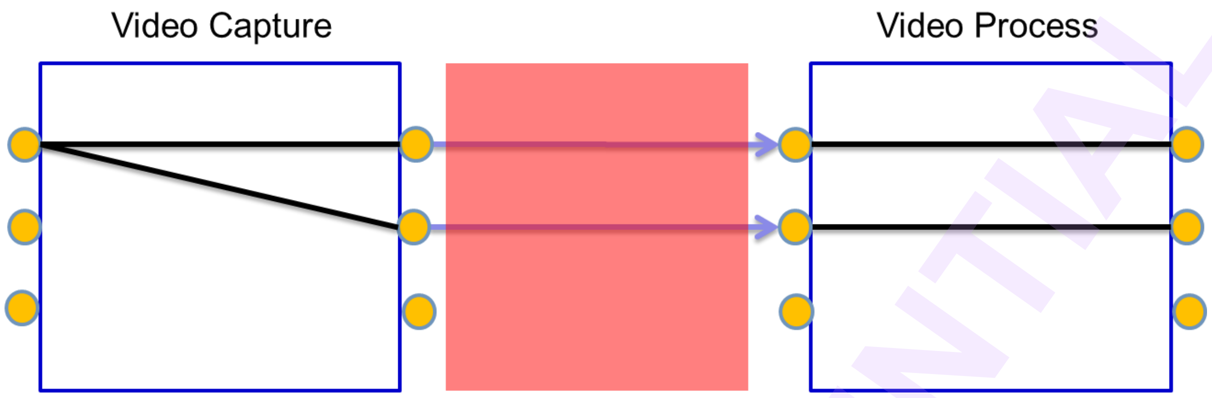

Fig. 5 A binding example between video capture and video process instances

```
     ret = hd_common_init(1);
    if(ret != HD_OK) {
        printf("init fail\n");
        goto exit;
    }

    ret = init_module();
    if(ret != HD_OK) {
        printf("init fail\n");
        goto exit;
    }

    ret = open_module();
    if(ret != HD_OK) {
        printf("open fail\n");
        goto exit;
    }

    //init dim for main/sub streams
     hd_videocap_get(cap0_path_id, HD_VIDEOCAP_PARAM_SYSCAPS, &cap_syscaps);

    main_dim.w  = cap_syscaps.max_dim.w;
    main_dim.h  = cap_syscaps.max_dim.h;
    sub_dim.w  = cap_syscaps.max_dim.w / 2;
    sub_dim.h  = cap_syscaps.max_dim.h / 2;

    ret = set_param();
    if(ret != HD_OK) {
        printf("set param fail\n");
        goto exit;
    }

    //bind main stream: VIDEOIN(0) -> VIDEOENC(0)
     hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOENC_0_IN_0);

     if (item == 0) {
        //bind sub stream: VIDEOIN(1) -> VIDEOENC(1)
          hd_videocap_bind(HD_VIDEOCAP_0_OUT_1, HD_VIDEOENC_0_IN_1);
     } else {
          //bind sub stream: VIDEOIN(0) -> VIDEOPROC(0) -> VIDEOENC(1)
          hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOPROC_0_IN_0);
          hd_videoproc_bind(HD_VIDEOPROC_0_OUT_0, HD_VIDEOENC_0_IN_1);
     }

     //let it start
     hd_videocap_start(cap0_path_id);
     hd_videoenc_start(enc0_path_id);
```

3.5. An example of stream instance connection

Fig 3 below shows a typical use cases of streaming instance, live view, playback, recording with audio encoding and decoding

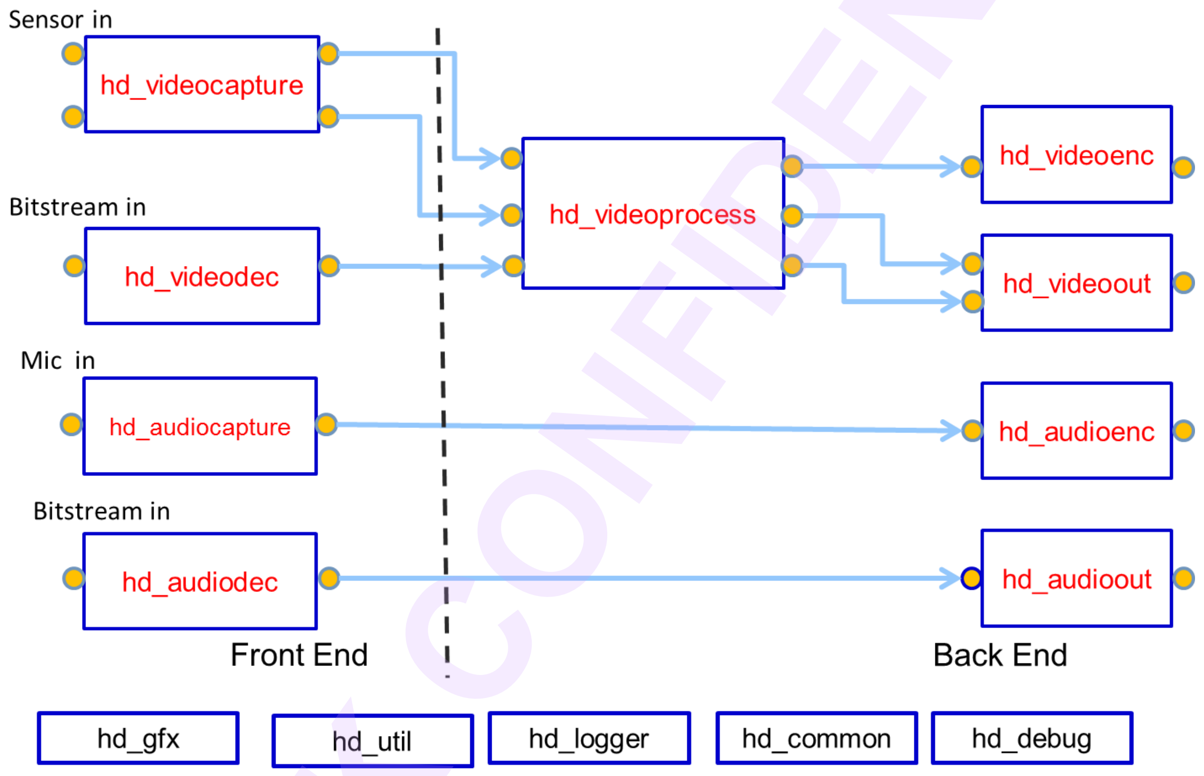

Fig 6 A typical use cases of streaming instance, live view, playback, recording with audio encoding and decoding

4. Source Connection

4.1. Overview

HDAL instances do several connecting work following stream path of each input. In this part, the connection process of sources like live view, playback, external input will be explained using digram and source code.

4.2. Live View Connection

The following figure shows the three stages in this liveview example code. In the first stage, the APIs *hd_xxx_open()* is used to open the instance path. The *hd_xxx_bind()* is to connect each instance path together in the second stage. After changing the status of each instance path to “start”, sensor data will be transmitted and processed to LCD or HDMI to implement the liveview function.

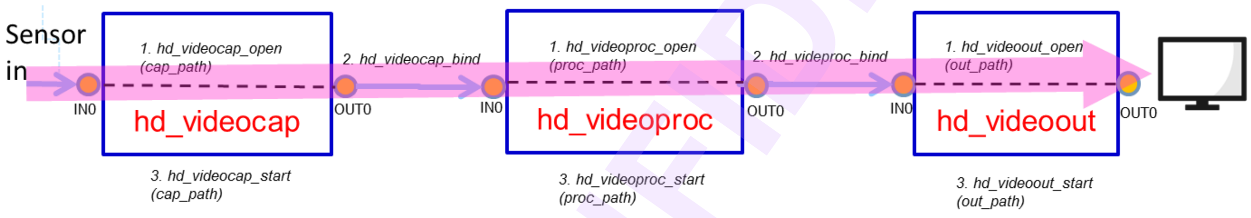
Sensor in
1. hd_videocap_open (cap_path)
IN0
hd_videocap
OUT0
3. hd_videocap_start (cap_path)
2. hd_videocap_bind
IN0
1. hd_videoproc_open (proc_path)
hd_videoproc
OUT0
3. hd_videoproc_start (proc_path)
2. hd_videproc_bind
IN0
1. hd_videoout_open (out_path)
hd_videoout
OUT0
3. hd_videoout_start (out_path)

Fig. 7 HDAL Liveview Connect Diagram

The following is the source code of this liveview example:

```
/**
	@brief Sample code of video liveview.\n

	@file video_liveview.c

	@author Janice Huang

	@ingroup mhdal

	@note Nothing.

	Copyright Novatek Microelectronics Corp. 2018.  All rights reserved.
*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>
#include "hdal.h"
#include "hd_debug.h"


#define DEBUG_MENU          1

#define CHKPNT              printf("\033[37mCHK: %s, %s: %d\033[0m\r\n",__FILE__,__func__,__LINE__)
#define DBGH(x)             printf("\033[0;35m%s=0x%08X\033[0m\r\n", #x, x)
#define DBGD(x)             printf("\033[0;35m%s=%d\033[0m\r\n", #x, x)

///////////////////////////////////////////////////////////////////////////////

#define RAW_COMPRESS_RATIO 70

#define DBGINFO_BUFSIZE()    (0x200)

//RAW
```

```
#define VDO_RAW_BUFSIZE(w, h, pxlfmt)   (ALIGN_CEIL_4((w) * HD_VIDEO_PXLFMT_BPP(pxlfmt) / 8) * (h))
//RAW compress only support 12bit mode
#define VDO_NRX_BUFSIZE(w, h)          ALIGN_CEIL_4(ALIGN_CEIL_4(ALIGN_CEIL_4((w) * 12 / 8) *
RAW_COMPRESS_RATIO / 100) * (h) + (ALIGN_CEIL_32(w) / 32) * 16 / 8)
//CA for AWB
#define VDO_CA_BUF_SIZE(win_num_w, win_num_h) ((win_num_w * win_num_h << 3) << 1)
//LA for AE
#define VDO_LA_BUF_SIZE(win_num_w, win_num_h) ((win_num_w * win_num_h << 1) << 1)
//YOUT for SHDR/WDR
#define VDO_YOUT_BUF_SIZE(win_num_w, win_num_h) (ALIGN_CEIL_4(win_num_w * 12 / 8) * win_num_h)
//ETH
#define VDO_ETH_BUF_SIZE(roi_w, roi_h, b_out_sel, b_8bit_sel) ((((roi_w >> (b_out_sel ? 1 : 0)) >>
(b_8bit_sel ? 0 : 2)) * (roi_h - 4)) >> (b_out_sel ? 1 : 0))
//VA for AF
#define VDO_VA_BUF_SIZE(win_num_w, win_num_h) ((win_num_w * win_num_h << 1) << 2)

#define VDO_YUV_BUFSIZE(w, h, pxlfmt)   ALIGN_CEIL_4(((w) * (h) * HD_VIDEO_PXLFMT_BPP(pxlfmt)) / 8)
#define VDO_NVX_BUFSIZE(w, h, pxlfmt)   (VDO_YUV_BUFSIZE(w, h, pxlfmt) * RAW_COMPRESS_RATIO / 100)

#define SEN_OUT_FMT          HD_VIDEO_PXLFMT_RAW12
#define CAP_OUT_FMT          HD_VIDEO_PXLFMT_RAW12
#define CA_WIN_NUM_W         32
#define CA_WIN_NUM_H         32
#define LA_WIN_NUM_W         32
#define LA_WIN_NUM_H         32
#define VA_WIN_NUM_W         16
#define VA_WIN_NUM_H         16
#define YOUT_WIN_NUM_W 128
#define YOUT_WIN_NUM_H 128
#define ETH_8BIT_SEL         0 //0: 2bit out, 1:8 bit out
#define ETH_OUT_SEL          1 //0: full, 1: subsample 1/2

#define VDO_SIZE_W           1920
#define VDO_SIZE_H           1080

///////////////////////////////////////////////////////////////////////////////


HD_RESULT mem_init(void)
{
      HD_RESULT             ret;
      HD_COMMON_MEM_INIT_CONFIG mem_cfg = {0};

      // config common pool (cap)
      mem_cfg.pool_info[0].type = HD_COMMON_MEM_COMMON_POOL;
      mem_cfg.pool_info[0].blk_size = DBGINFO_BUFSIZE()+VDO_RAW_BUFSIZE(VDO_SIZE_W, VDO_SIZE_H,
CAP_OUT_FMT)

      +VDO_CA_BUF_SIZE(CA_WIN_NUM_W, CA_WIN_NUM_H)

      +VDO_LA_BUF_SIZE(LA_WIN_NUM_W, LA_WIN_NUM_H)

      +VDO_YOUT_BUF_SIZE(YOUT_WIN_NUM_W, YOUT_WIN_NUM_H)

      +VDO_ETH_BUF_SIZE(VDO_SIZE_W, VDO_SIZE_H, ETH_OUT_SEL, ETH_8BIT_SEL);
      mem_cfg.pool_info[0].blk_cnt = 3;
      mem_cfg.pool_info[0].ddr_id = DDR_ID0;
      // config common pool (main)
      mem_cfg.pool_info[1].type = HD_COMMON_MEM_COMMON_POOL;
      mem_cfg.pool_info[1].blk_size = DBGINFO_BUFSIZE()+VDO_YUV_BUFSIZE(VDO_SIZE_W, VDO_SIZE_H,
HD_VIDEO_PXLFMT_YUV420);
      mem_cfg.pool_info[1].blk_cnt = 3;
      mem_cfg.pool_info[1].ddr_id = DDR_ID0;

      ret = hd_common_mem_init(&mem_cfg);
      return ret;
}

HD_RESULT mem_exit(void)
{
      HD_RESULT ret = HD_OK;
      hd_common_mem_uninit();
```

```
	return ret;
}

//////////////////////////////////////////////////////////////////////////////////

HD_RESULT get_cap_caps(HD_PATH_ID video_cap_ctrl, HD_VIDEOCAP_SYSCAPS *p_video_cap_syscaps)
{
	HD_RESULT ret = HD_OK;
	hd_videocap_get(video_cap_ctrl, HD_VIDEOCAP_PARAM_SYSCAPS, p_video_cap_syscaps);
	return ret;
}

HD_RESULT get_cap_sysinfo(HD_PATH_ID video_cap_ctrl)
{
	HD_RESULT ret = HD_OK;
	HD_VIDEOCAP_SYSINFO sys_info = {0};

	hd_videocap_get(video_cap_ctrl, HD_VIDEOCAP_PARAM_SYSINFO, &sys_info);
	printf("sys_info.devid =0x%X, cur_fps[0]=%d/%d, vd_count=%llu\r\n", sys_info.dev_id,
GET_HI_UINT16(sys_info.cur_fps[0]), GET_LO_UINT16(sys_info.cur_fps[0]), sys_info.vd_count);
	return ret;
}

HD_RESULT set_cap_cfg(HD_PATH_ID *p_video_cap_ctrl)
{
	HD_RESULT ret = HD_OK;
	HD_VIDEOCAP_DRV_CONFIG cap_cfg = {0};
	HD_PATH_ID video_cap_ctrl = 0;
	HD_VIDEOCAP_CTRL iq_ctl = {0};

	snprintf(cap_cfg.sen_cfg.sen_dev.driver_name, HD_VIDEOCAP_SEN_NAME_LEN-1, "nvt_sen_imx291");
	cap_cfg.sen_cfg.sen_dev.if_type = HD_COMMON_VIDEO_IN_MIPI_CSI;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.pinmux.sensor_pinmux =  0x220; //PIN_SENSOR_CFG_MIPI |
PIN_SENSOR_CFG_MCLK
	cap_cfg.sen_cfg.sen_dev.pin_cfg.pinmux.serial_if_pinmux = 0xf04;//PIN_MIPI_LVDS_CFG_CLK2 |
PIN_MIPI_LVDS_CFG_DAT0|PIN_MIPI_LVDS_CFG_DAT1 | PIN_MIPI_LVDS_CFG_DAT2 | PIN_MIPI_LVDS_CFG_DAT3
	cap_cfg.sen_cfg.sen_dev.pin_cfg.pinmux.cmd_if_pinmux = 0x10;//PIN_I2C_CFG_CH2
	cap_cfg.sen_cfg.sen_dev.pin_cfg.clk_lane_sel = HD_VIDEOCAP_SEN_CLANE_SEL_CSI0_USE_C2;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[0] = 0;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[1] = 1;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[2] = HD_VIDEOCAP_SEN_IGNORE;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[3] = HD_VIDEOCAP_SEN_IGNORE;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[4] = HD_VIDEOCAP_SEN_IGNORE;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[5] = HD_VIDEOCAP_SEN_IGNORE;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[6] = HD_VIDEOCAP_SEN_IGNORE;
	cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[7] = HD_VIDEOCAP_SEN_IGNORE;
	ret = hd_videocap_open(0, HD_VIDEOCAP_0_CTRL, &video_cap_ctrl); //open this for device control
	if (ret != HD_OK) {
		return ret;
	}
	ret |= hd_videocap_set(video_cap_ctrl, HD_VIDEOCAP_PARAM_DRV_CONFIG, &cap_cfg);
	iq_ctl.func = HD_VIDEOCAP_FUNC_AE | HD_VIDEOCAP_FUNC_AWB;
	ret |= hd_videocap_set(video_cap_ctrl, HD_VIDEOCAP_PARAM_CTRL, &iq_ctl);

	*p_video_cap_ctrl = video_cap_ctrl;
	return ret;
}

HD_RESULT set_cap_param(HD_PATH_ID video_cap_path, HD_DIM *p_dim)
{
	HD_RESULT ret = HD_OK;
	{//select sensor mode, manually or automatically
		HD_VIDEOCAP_IN video_in_param = {0};

		video_in_param.sen_mode = HD_VIDEOCAP_SEN_MODE_AUTO; //auto select sensor mode by the parameter
of HD_VIDEOCAP_PARAM_OUT
		video_in_param.frc = HD_VIDEO_FRC_RATIO(30,1);
		video_in_param.dim.w = p_dim->w;
		video_in_param.dim.h = p_dim->h;
		video_in_param.pxlfmt = SEN_OUT_FMT;
		video_in_param.out_frame_num = HD_VIDEOCAP_SEN_FRAME_NUM_1;
		ret = hd_videocap_set(video_cap_path, HD_VIDEOCAP_PARAM_IN, &video_in_param);
```

```
            //printf("set_cap_param MODE=%d\r\n", ret);
            if (ret != HD_OK) {
                return ret;
            }
        }
        #if 1 //no crop, full frame
        {
            HD_VIDEOCAP_CROP video_crop_param = {0};

            video_crop_param.mode = HD_CROP_OFF;
            ret = hd_videocap_set(video_cap_path, HD_VIDEOCAP_PARAM_OUT_CROP, &video_crop_param);
            //printf("set_cap_param CROP NONE=%d\r\n", ret);
        }
        #else //HD_CROP_ON
        {
            HD_VIDEOCAP_CROP video_crop_param = {0};

            video_crop_param.mode = HD_CROP_ON;
            video_crop_param.win.rect.x = 0;
            video_crop_param.win.rect.y = 0;
            video_crop_param.win.rect.w = 1920/2;
            video_crop_param.win.rect.h= 1080/2;
            video_crop_param.align.w = 4;
            video_crop_param.align.h = 4;
            ret = hd_videocap_set(video_cap_path, HD_VIDEOCAP_PARAM_OUT_CROP, &video_crop_param);
            //printf("set_cap_param CROP ON=%d\r\n", ret);
        }
        #endif
        {
            HD_VIDEOCAP_OUT video_out_param = {0};

            //without setting dim for no scaling, using original sensor out size
            video_out_param.pxlfmt = CAP_OUT_FMT;
            video_out_param.dir = HD_VIDEO_DIR_NONE;
            ret = hd_videocap_set(video_cap_path, HD_VIDEOCAP_PARAM_OUT, &video_out_param);
            //printf("set_cap_param OUT=%d\r\n", ret);
        }

        return ret;
}

///////////////////////////////////////////////////////////////////////////////

HD_RESULT set_proc_cfg(HD_PATH_ID *p_video_proc_ctrl, HD_DIM* p_max_dim)
{
        HD_RESULT ret = HD_OK;
        HD_VIDEOPROC_DEV_CONFIG video_cfg_param = {0};
        HD_VIDEOPROC_CTRL video_ctrl_param = {0};
        HD_PATH_ID video_proc_ctrl = 0;

        ret = hd_videoproc_open(0, HD_VIDEOPROC_0_CTRL, &video_proc_ctrl); //open this for device control
        if (ret != HD_OK)
            return ret;

        if (p_max_dim != NULL ) {
            video_cfg_param.pipe = HD_VIDEOPROC_PIPE_RAWALL;
            video_cfg_param.isp_id = 0;
            video_cfg_param.ctrl_max.func = 0;
            video_cfg_param.in_max.func = 0;
            video_cfg_param.in_max.dim.w = p_max_dim->w;
            video_cfg_param.in_max.dim.h = p_max_dim->h;
            video_cfg_param.in_max.pxlfmt = CAP_OUT_FMT;
            video_cfg_param.in_max.frc = HD_VIDEO_FRC_RATIO(1,1);
            ret = hd_videoproc_set(video_proc_ctrl, HD_VIDEOPROC_PARAM_DEV_CONFIG, &video_cfg_param);
            if (ret != HD_OK) {
                return HD_ERR_NG;
            }
        }

        video_ctrl_param.func = 0;
        ret = hd_videoproc_set(video_proc_ctrl, HD_VIDEOPROC_PARAM_CTRL, &video_ctrl_param);
```

```
	*p_video_proc_ctrl = video_proc_ctrl;

	return ret;
}

HD_RESULT set_proc_param(HD_PATH_ID video_proc_path, HD_DIM* p_dim)
{
	HD_RESULT ret = HD_OK;

	if (p_dim != NULL) { //if videoproc is already binding to dest module, not require to setting this!
		HD_VIDEOPROC_OUT video_out_param = {0};
		video_out_param.func = 0;
		video_out_param.dim.w = p_dim->w;
		video_out_param.dim.h = p_dim->h;
		video_out_param.pxlfmt = HD_VIDEO_PXLFMT_YUV420;
		video_out_param.dir = HD_VIDEO_DIR_NONE;
		video_out_param.frc = HD_VIDEO_FRC_RATIO(1,1);
		ret = hd_videoproc_set(video_proc_path, HD_VIDEOPROC_PARAM_OUT, &video_out_param);
	}

	return ret;
}

///////////////////////////////////////////////////////////////////////////////

HD_RESULT set_out_cfg(HD_PATH_ID *p_video_out_ctrl, UINT32 out_type)
{
	HD_RESULT ret = HD_OK;
	HD_VIDEOOUT_MODE videoout_mode = {0};
	HD_PATH_ID video_out_ctrl = 0;

	ret = hd_videoout_open(0, HD_VIDEOOUT_0_CTRL, &video_out_ctrl); //open this for device control
	if (ret != HD_OK) {
		return ret;
	}

	printf("out_type=%d\r\n", out_type);
	switch(out_type){
	case 0:
		videoout_mode.output_type = HD_COMMON_VIDEO_OUT_CVBS;
		videoout_mode.input_dim = HD_VIDEOOUT_IN_AUTO;
		videoout_mode.output_mode.cvbs= HD_VIDEOOUT_CVBS_NTSC;
	break;
	case 1:
		videoout_mode.output_type = HD_COMMON_VIDEO_OUT_LCD;
		videoout_mode.input_dim = HD_VIDEOOUT_IN_AUTO;
		videoout_mode.output_mode.lcd = HD_VIDEOOUT_LCD_0;
	break;
	case 2:
		videoout_mode.output_type = HD_COMMON_VIDEO_OUT_HDMI;
		videoout_mode.input_dim = HD_VIDEOOUT_IN_AUTO;
		videoout_mode.output_mode.hdmi= HD_VIDEOOUT_HDMI_1920X1080P30;
	break;
	default:
		printf("not support out_type\r\n");
	break;
	}
	ret = hd_videoout_set(video_out_ctrl, HD_VIDEOOUT_PARAM_MODE, &videoout_mode);

	*p_video_out_ctrl=video_out_ctrl ;
	return ret;
}

HD_RESULT get_out_caps(HD_PATH_ID video_out_ctrl,HD_VIDEOOUT_SYSCAPS *p_video_out_syscaps)
{
	HD_RESULT ret = HD_OK;
    HD_DEVCOUNT video_out_dev = {0};

	ret = hd_videoout_get(video_out_ctrl, HD_VIDEOOUT_PARAM_DEVCOUNT, &video_out_dev);
	if (ret != HD_OK) {
		return ret;
	}
	printf("##devcount %d\r\n", video_out_dev.max_dev_count);

	ret = hd_videoout_get(video_out_ctrl, HD_VIDEOOUT_PARAM_SYSCAPS, p_video_out_syscaps);
	if (ret != HD_OK) {
		return ret;
	}
	return ret;
}

HD_RESULT set_out_param(HD_PATH_ID video_out_path, HD_DIM *p_dim)
{
	HD_RESULT ret = HD_OK;
	HD_VIDEOOUT_IN video_out_param={0};

	video_out_param.dim.w = p_dim->w;
	video_out_param.dim.h = p_dim->h;
	video_out_param.pxlfmt = HD_VIDEO_PXLFMT_YUV420;
	video_out_param.dir = HD_VIDEO_DIR_NONE;
	ret = hd_videoout_set(video_out_path, HD_VIDEOOUT_PARAM_IN, &video_out_param);
	if (ret != HD_OK) {
		return ret;
	}
	memset((void *)&video_out_param,0,sizeof(HD_VIDEOOUT_IN));
	ret = hd_videoout_get(video_out_path, HD_VIDEOOUT_PARAM_IN, &video_out_param);
	if (ret != HD_OK) {
		return ret;
	}
	printf("##video_out_param w:%d,h:%d %x %x\r\n", video_out_param.dim.w, video_out_param.dim.h,
video_out_param.pxlfmt, video_out_param.dir);

	return ret;
}

///////////////////////////////////////////////////////////////////////////////

typedef struct _VIDEO_LIVEVIEW {

	// (1)
	HD_VIDEOCAP_SYSCAPS cap_syscaps;
	HD_PATH_ID cap_ctrl;
	HD_PATH_ID cap_path;

	HD_DIM  cap_dim;
	HD_DIM  proc_max_dim;

	// (2)
	HD_VIDEOPROC_SYSCAPS proc_syscaps;
	HD_PATH_ID proc_ctrl;
	HD_PATH_ID proc_path;

	HD_DIM  out_max_dim;
	HD_DIM  out_dim;

	// (3)
	HD_VIDEOOUT_SYSCAPS out_syscaps;
	HD_PATH_ID out_ctrl;
	HD_PATH_ID out_path;

} VIDEO_LIVEVIEW;

HD_RESULT init_module(void)
{
	HD_RESULT ret;
	if ((ret = hd_videocap_init()) != HD_OK)
		return ret;
	if ((ret = hd_videoproc_init()) != HD_OK)
		return ret;
	if ((ret = hd_videoout_init()) != HD_OK)
		return ret;
	return HD_OK;
}
```

```
HD_RESULT open_module(VIDEO_LIVEVIEW *p_stream, HD_DIM* p_proc_max_dim, UINT32 out_type)
{
      HD_RESULT ret;
      // set videocap config
      ret = set_cap_cfg(&p_stream->cap_ctrl);
      if (ret != HD_OK) {
            printf("set cap-cfg fail=%d\n", ret);
            return HD_ERR_NG;
      }
      // set videoproc config
      ret = set_proc_cfg(&p_stream->proc_ctrl, p_proc_max_dim);
      if (ret != HD_OK) {
            printf("set proc-cfg fail=%d\n", ret);
            return HD_ERR_NG;
      }
      // set videoout config
      ret = set_out_cfg(&p_stream->out_ctrl, out_type);
      if (ret != HD_OK) {
            printf("set out-cfg fail=%d\n", ret);
            return HD_ERR_NG;
      }
      if ((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_0, &p_stream->cap_path)) != HD_OK)
            return ret;
      if ((ret = hd_videoproc_open(HD_VIDEOPROC_0_IN_0, HD_VIDEOPROC_0_OUT_0, &p_stream->proc_path)) !=
HD_OK)
            return ret;
      if ((ret = hd_videoout_open(HD_VIDEOOUT_0_IN_0, HD_VIDEOOUT_0_OUT_0, &p_stream->out_path)) != HD_OK)
            return ret;

      return HD_OK;
}

HD_RESULT close_module(VIDEO_LIVEVIEW *p_stream)
{
      HD_RESULT ret;
      if ((ret = hd_videocap_close(p_stream->cap_path)) != HD_OK)
            return ret;
      if ((ret = hd_videoproc_close(p_stream->proc_path)) != HD_OK)
            return ret;
      if ((ret = hd_videoout_close(p_stream->out_path)) != HD_OK)
            return ret;
      return HD_OK;
}

HD_RESULT exit_module(void)
{
      HD_RESULT ret;
      if ((ret = hd_videocap_uninit()) != HD_OK)
            return ret;
      if ((ret = hd_videoproc_uninit()) != HD_OK)
            return ret;
      if ((ret = hd_videoout_uninit()) != HD_OK)
            return ret;
      return HD_OK;
}


int main(int argc, char** argv)
{
      HD_RESULT ret;
      INT key;
      VIDEO_LIVEVIEW stream[1] = {0}; //0: main stream
      UINT32 out_type = 0;

      // query program options
      if (argc == 2) {
            out_type = atoi(argv[1]);
            printf("out_type %d\r\n", out_type);
            if(out_type > 2) {
                  printf("error: not support out_type!\r\n");
                  return 0;
            }
```

```
    }

    // init hdal
    ret = hd_common_init(0);
    if (ret != HD_OK) {
        printf("common fail=%d\n", ret);
        goto exit;
    }

    // init memory
    ret = mem_init();
    if (ret != HD_OK) {
        printf("mem fail=%d\n", ret);
        goto exit;
    }

    // init all modules
    ret = init_module();
    if (ret != HD_OK) {
        printf("init fail=%d\n", ret);
        goto exit;
    }

    // open video_stream modules (main)
    stream[0].proc_max_dim.w = VDO_SIZE_W; //assign by user
    stream[0].proc_max_dim.h = VDO_SIZE_H; //assign by user
    ret = open_module(&stream[0], &stream[0].proc_max_dim, out_type);
    if (ret != HD_OK) {
        printf("open fail=%d\n", ret);
        goto exit;
    }

    // get videocap capability
    ret = get_cap_caps(stream[0].cap_ctrl, &stream[0].cap_syscaps);
    if (ret != HD_OK) {
        printf("get cap-caps fail=%d\n", ret);
        goto exit;
    }

    // get videoout capability
    ret = get_out_caps(stream[0].out_ctrl, &stream[0].out_syscaps);
    if (ret != HD_OK) {
        printf("get out-caps fail=%d\n", ret);
        goto exit;
    }
    stream[0].out_max_dim = stream[0].out_syscaps.output_dim;

    // set videocap parameter
    stream[0].cap_dim.w = VDO_SIZE_W; //assign by user
    stream[0].cap_dim.h = VDO_SIZE_H; //assign by user
    ret = set_cap_param(stream[0].cap_path, &stream[0].cap_dim);
    if (ret != HD_OK) {
        printf("set cap fail=%d\n", ret);
        goto exit;
    }

    // set videoproc parameter (main)
    ret = set_proc_param(stream[0].proc_path, NULL);
    if (ret != HD_OK) {
        printf("set proc fail=%d\n", ret);
        goto exit;
    }

    // set videoout parameter (main)
    stream[0].out_dim.w = stream[0].out_max_dim.w; //using device max dim.w
    stream[0].out_dim.h = stream[0].out_max_dim.h; //using device max dim.h
    ret = set_out_param(stream[0].out_path, &stream[0].out_dim);
    if (ret != HD_OK) {
        printf("set out fail=%d\n", ret);
        goto exit;
    }
```

```
	// bind video_liveview modules (main)
	hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOPROC_0_IN_0);
	hd_videoproc_bind(HD_VIDEOPROC_0_OUT_0, HD_VIDEOOUT_0_IN_0);

	// start video_liveview modules (main)
	hd_videocap_start(stream[0].cap_path);
	hd_videoproc_start(stream[0].proc_path);
	// wait ae/awb stable
	sleep(1);
	hd_videoout_start(stream[0].out_path);

	// query user key
	printf("Enter q to exit\n");
	while (1) {
		key = getchar();
		if (key == 'q' || key == 0x3) {
			// quit program
			break;
		}

		#if (DEBUG_MENU == 1)
		if (key == 'd') {
			// enter debug menu
			hd_debug_run_menu();
			printf("\r\nEnter q to exit, Enter d to debug\r\n");
		}
		#endif
	}

	// stop video_liveview modules (main)
	hd_videocap_stop(stream[0].cap_path);
	hd_videoproc_stop(stream[0].proc_path);
	hd_videoout_stop(stream[0].out_path);

	// unbind video_liveview modules (main)
	hd_videocap_unbind(HD_VIDEOCAP_0_OUT_0);
	hd_videoproc_unbind(HD_VIDEOPROC_0_OUT_0);

exit:
	// close video_liveview modules (main)
	ret = close_module(&stream[0]);
	if (ret != HD_OK) {
		printf("close fail=%d\n", ret);
	}

	// uninit all modules
	ret = exit_module();
	if (ret != HD_OK) {
		printf("exit fail=%d\n", ret);
	}

	// uninit memory
	ret = mem_exit();
	if (ret != HD_OK) {
		printf("mem fail=%d\n", ret);
	}

	// uninit hdal
	ret = hd_common_uninit();
	if (ret != HD_OK) {
		printf("common fail=%d\n", ret);
	}

	return 0;
}
```

This example code flow is described as the following steps:

1. Initialize the modules used in Liveview: hd_videocapture, hd_videoprocess and hd_videoout.
2. Create the capture path, the video process path and the output path.
3. Get the video capability, such as the videocap capability and the LCD resolution.

4. Set the parameters of the capture path, such as the frame width and the frame height. And set the parameters of the output path, such as the output position (x, y) and the dimension (width, height).
5. Connect the paths, the stream flow is built by the following connections:
   hd_videodec_bind(HD_VIDEOCAP_OUT(0,0), HD_VIDEOPROC_IN(0,0));
   The outport #0 of the capture path connectes to the inport #0 of the video process path.
   hd_videoproc_bind(HD_VIDEOPROC_OUT(0,0), HD_VIDEOOUT_IN(0,0));
   The outport #0 of the video process path connectes to the inport #0 of the output path.
6. The stream flow starts running after changing the status of the paths, including the capture path, the video process path and the output path to "Start".
7. Check the key button in the console window. In this sample code, we use "d" key to enter the debug menu.
8. Press "q" key to close this sample code.
9. Change the status of the capture path, the video process path and the output path to "Stop".
10. Disconnect all connections.
11. Close all instance paths.
12. Uninitialize all modules.

4.3. Playback Connection

The following figure shows the three stages in this playback example code. In the first stage, the APIs *hd_xxx_open()* is used to open the instance path. The *hd_xxx_bind()* is to connect each instance path together in the second stage. After changing the status of each instance path to "run", the API *hd_videodec_send_list()* is used to send the h.264 bit stream to the decode instance path for decoding.

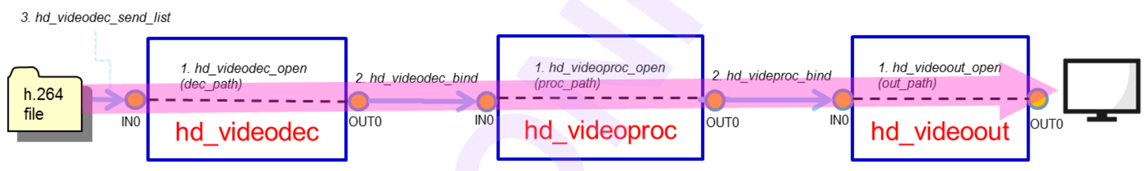

Fig. 8 HDAL Playback Connect Diagram

The following is the source code of this playback example:

```
/**
 * @file playback_speed.c
 * @brief playback speed adjustment.
 * @author Foster Huang
 * @date in the year 2018
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <asm/ioctl.h>
#include "hdal.h"

#define LCD_PLAY_TIMEVAL   33333           // 33333 microseconds

typedef struct {                          // playback speed control
   char *speed_str;                       // message
   unsigned int numerator;                // speed numerator
   unsigned int denominator;              // speed denominator
} pb_speed_t;

typedef struct {                          // video path
   HD_VIDEOOUT_SYSCAPS out_syscaps;       // video out capibility
   HD_PATH_ID out_ctrl;                   // video out control
   HD_PATH_ID out_path;                   // video out path
   HD_PATH_ID dec_path;                   // video decode path
   HD_PATH_ID proc_path;                  // video process path
   HD_VIDEOOUT_SYSCAPS lcd_syscaps;        // lcd capibility
} pb_video_t;

typedef struct {                           // speed control
   pb_speed_t pb_speed[7];                 // speed control table
   unsigned int speed_level;               // speed control level for 1x, 2x, ...
   unsigned int pb_pause;                  // puase indicator
   unsigned int pb_exit;                   // exit indictor
} pb_speed_ctrl_t;

typedef struct {                           // sample pattern
   char name[40];                         // file name
   int frame_rate;                        // frame rate
   int img_width;                         // image width
   int img_height;                        // image height
} pb_pattern_t;

typedef struct {                           // sample program data structure
   pb_video_t video_path;
   pb_speed_ctrl_t speed_ctrl;
   pb_pattern_t pattern;
} pb_speed_demo_t;

unsigned int time_to_ms(struct timeval *a)
{
   return (a->tv_sec * 1000) + (a->tv_usec / 1000);
}

static void *playback_thread(void *arg)
{
   int ret = 0, length = 0;
   char filename[40];
   FILE *bs_fd, *len_fd;
   char *data;
   unsigned int frame_period;
   unsigned int playback_time;
   unsigned int current_ms;
   HD_VIDEODEC_SEND_LIST video_bs;
   struct timeval time_start;
   unsigned int diff, sleep_ms = 0;
   unsigned int speed_level, speed_numerator, speed_denominator;
   pb_speed_demo_t *p_pb_speed_demo = (pb_speed_demo_t *)arg;
   pb_pattern_t *p_pattern = &(p_pb_speed_demo->pattern);
   pb_speed_ctrl_t *p_speed_ctrl = &(p_pb_speed_demo->speed_ctrl);
   pb_video_t *p_video_path = &(p_pb_speed_demo->video_path);

   snprintf(filename, sizeof(filename), "%s.265", p_pattern->name);
   if ((bs_fd = fopen(filename, "rb")) == NULL) {
      printf("[ERROR] Open %s failed!!\n", filename);
      exit(1);
   }
   snprintf(filename, sizeof(filename), "%s.len", p_pattern->name);
   if ((len_fd = fopen(filename, "rb")) == NULL) {
      printf("[ERROR] Open %s failed!!\n", filename);
      exit(1);
   }
   //data = (char *)malloc(BITSTREAM_LEN);
   data = (char *)malloc(p_pattern->img_width * p_pattern->img_height * 3 / 2);
   if (!data) {
      printf("Error allocation\n");
      exit(1);
   }
   frame_period = 1000 / p_pattern->frame_rate;
   printf("Playback pattern %s.264 (frame time %dms) starting..\n", p_pattern->name, frame_period);

   gettimeofday(&time_start, NULL);
   current_ms = time_to_ms(&time_start);
   playback_time = current_ms + frame_period;

   while (1) {
      if (p_speed_ctrl->pb_exit == 1) {
          break;
      }
      if (p_speed_ctrl->pb_pause == 1) {
          sleep(1);
          continue;
      }

      gettimeofday(&time_start, NULL);
      current_ms = time_to_ms(&time_start);

      memset(&video_bs, 0, sizeof(video_bs));  //clear all video_bs
      sleep_ms = 0;
      if (playback_time > current_ms) {
          diff = (playback_time - current_ms);
          if (diff > 66)
              sleep_ms = diff;
      }
      if (sleep_ms) {
          usleep(sleep_ms * 1000);    //no data playback
          continue;
      }
      if (fscanf(len_fd, "%d\n", &length) == EOF) {
          fseek(bs_fd, 0, SEEK_SET);
          fseek(len_fd, 0, SEEK_SET);
          fscanf(len_fd, "%d\n", &length);
      }
      if (length == 0) {
          continue;
      }
      fread(data, 1, length, bs_fd);
      video_bs.path_id = p_video_path->dec_path;
      video_bs.user_bs.p_bs_buf = data;
      video_bs.user_bs.bs_buf_size = length;
      video_bs.user_bs.time_align = HD_VIDEODEC_TIME_ALIGN_USER;
      speed_level = p_speed_ctrl->speed_level;
      speed_numerator = p_speed_ctrl->pb_speed[speed_level].numerator;
      speed_denominator = p_speed_ctrl->pb_speed[speed_level].denominator;
      video_bs.user_bs.time_diff = LCD_PLAY_TIMEVAL * speed_numerator / speed_denominator;
      if ((ret = hd_videodec_send_list(&video_bs, 1, 500)) < 0) {
          printf("<send bitstream fail(%d)!>\n", ret);
          continue;
      }
      playback_time += (frame_period * speed_numerator / speed_denominator);
   }

   fclose(bs_fd);
   fclose(len_fd);
   free(data);
   return 0;
}

HD_RESULT get_video_syscaps(pb_video_t *p_pb_video)
{
   HD_RESULT ret;

   ret = hd_videoout_get(p_pb_video->out_ctrl, HD_VIDEOOUT_PARAM_SYSCAPS, &(p_pb_video->lcd_syscaps));
   if (ret != HD_OK) {
      printf("hd_videoout_get:param_id(%d) video_out_ctrl(%#lx) fail\n", HD_VIDEOOUT_PARAM_SYSCAPS,
p_pb_video->out_ctrl);
   }
   return ret;
}

HD_RESULT set_video_cfg(pb_speed_demo_t *p_pb_speed_demo)
{
   HD_VIDEOOUT_WIN_ATTR win;
   HD_RESULT ret = HD_OK;
   HD_VIDEOPROC_CROP proc_crop;
   HD_VIDEOPROC_OUT proc_out;
   HD_VIDEODEC_PATH_CONFIG dec_config;
   HD_VIDEOPROC_DEV_CONFIG vpe_config;
   pb_pattern_t *p_pattern = &(p_pb_speed_demo->pattern);
   pb_video_t *p_video_path = &(p_pb_speed_demo->video_path);

   /* Set videodec parameters */
   memset(&dec_config, 0x0, sizeof(HD_VIDEODEC_PATH_CONFIG));
   dec_config.max_mem.dim.w = p_pattern->img_width;
   dec_config.max_mem.dim.h = p_pattern->img_height;
   dec_config.max_mem.frame_rate = p_pattern->frame_rate;
   dec_config.max_mem.bs_counts = 4;

   /* Set videodec out pool */
   dec_config.data_pool[0].mode = HD_VIDEODEC_POOL_ENABLE;
   dec_config.data_pool[0].ddr_id = 0;
   dec_config.data_pool[0].counts = HD_VIDEODEC_SET_COUNT(3, 0);
   dec_config.data_pool[0].max_counts = HD_VIDEODEC_SET_COUNT(3, 0);

   dec_config.data_pool[1].mode = HD_VIDEODEC_POOL_ENABLE;
   dec_config.data_pool[1].ddr_id = 0;
   dec_config.data_pool[1].counts = HD_VIDEODEC_SET_COUNT(3, 0);
   dec_config.data_pool[1].max_counts = HD_VIDEODEC_SET_COUNT(3, 0);

   dec_config.data_pool[2].mode = HD_VIDEODEC_POOL_DISABLE;
   dec_config.data_pool[3].mode = HD_VIDEODEC_POOL_DISABLE;

   ret = hd_videodec_set(p_video_path->dec_path, HD_VIDEODEC_PARAM_PATH_CONFIG, &dec_config);
   if (ret != HD_OK) {
      return ret;
   }

   /* Set videoprocess out pool */
   memset(&vpe_config, 0x0, sizeof(HD_VIDEOPROC_DEV_CONFIG));
   vpe_config.data_pool[0].mode = HD_VIDEOPROC_POOL_ENABLE;
   vpe_config.data_pool[0].ddr_id = 0;
   vpe_config.data_pool[0].counts = HD_VIDEOPROC_SET_COUNT(3, 0);
   vpe_config.data_pool[0].max_counts = HD_VIDEOPROC_SET_COUNT(3, 0);

   vpe_config.data_pool[1].mode = HD_VIDEOPROC_POOL_DISABLE;
   vpe_config.data_pool[2].mode = HD_VIDEOPROC_POOL_DISABLE;
   vpe_config.data_pool[3].mode = HD_VIDEOPROC_POOL_DISABLE;

   ret = hd_videoproc_set(p_video_path->proc_path, HD_VIDEOPROC_PARAM_DEV_CONFIG, &vpe_config);
   if (ret != HD_OK) {
      return ret;
   }

   /* Set videoproc output setting */
   proc_crop.win.rect.x = 0;
   proc_crop.win.rect.y = 0;
   proc_crop.win.rect.w = p_video_path->lcd_syscaps.input_dim.w;
   proc_crop.win.rect.h = p_video_path->lcd_syscaps.input_dim.h;
   proc_crop.win.coord.w = p_video_path->lcd_syscaps.input_dim.w;
   proc_crop.win.coord.h = p_video_path->lcd_syscaps.input_dim.h;
   ret = hd_videoproc_set(p_video_path->proc_path, HD_VIDEOPROC_PARAM_OUT_CROP, &proc_crop);
   if (ret != HD_OK) {
      return ret;
   }
   proc_out.pxlfmt = HD_VIDEO_PXLFMT_YUV422_ONE;
   proc_out.dir = HD_VIDEO_DIR_NONE;
   ret = hd_videoproc_set(p_video_path->proc_path, HD_VIDEOPROC_PARAM_OUT, &proc_out);
   if (ret != HD_OK) {
      return ret;
   }

   /* Set videoout input setting */
   win.rect.x = 0;
   win.rect.y = 0;
   win.rect.w = p_video_path->lcd_syscaps.input_dim.w;
   win.rect.h = p_video_path->lcd_syscaps.input_dim.h;
   win.visible = 1;
   ret = hd_videoout_set(p_video_path->out_path, HD_VIDEOOUT_PARAM_IN_WIN_ATTR, &win);
   if (ret != HD_OK) {
      return ret;
   }
```

```
   return ret;
}

HD_RESULT init_module(void)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videodec_init()) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_init()) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoout_init()) != HD_OK) {
      return ret;
   }
   return ret;
}

HD_RESULT uninit_module(void)
{
   HD_RESULT ret = HD_OK;
   if ((ret = hd_videodec_uninit()) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_uninit()) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoout_uninit()) != HD_OK) {
      return ret;
   }
   return ret;
}

HD_RESULT open_module(pb_video_t *p_pb_video)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videoout_open(0, HD_VIDEOOUT_0_CTRL, &(p_pb_video->out_ctrl))) != HD_OK)   //open this for
device control
      return ret;
   if ((ret = hd_videodec_open(HD_VIDEODEC_IN(0, 0), HD_VIDEODEC_OUT(0, 0), &(p_pb_video->dec_path))) !=
HD_OK)
      return ret;
   if ((ret = hd_videoproc_open(HD_VIDEOPROC_IN(0, 0), HD_VIDEOPROC_OUT(0, 0),
&(p_pb_video->proc_path))) != HD_OK)
      return ret;
   if ((ret = hd_videoout_open(HD_VIDEOOUT_IN(0, 0), HD_VIDEOOUT_OUT(0, 0), &(p_pb_video->out_path))) !=
HD_OK)
      return ret;

   return ret;
}

HD_RESULT close_module(pb_video_t *p_pb_video)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videoout_close(p_pb_video->out_ctrl)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videodec_close(p_pb_video->dec_path)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_close(p_pb_video->proc_path)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoout_close(p_pb_video->out_path)) != HD_OK) {
      return ret;
   }
   return ret;
```

```
}

HD_RESULT bind_unit(void)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videodec_bind(HD_VIDEODEC_OUT(0, 0), HD_VIDEOPROC_IN(0, 0))) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_bind(HD_VIDEOPROC_OUT(0, 0), HD_VIDEOOUT_IN(0, 0))) != HD_OK) {
      return ret;
   }
   return ret;
}

HD_RESULT unbind_unit(void)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videodec_unbind(HD_VIDEODEC_OUT(0, 0))) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_unbind(HD_VIDEOPROC_OUT(0, 0))) != HD_OK) {
      return ret;
   }
   return ret;
}

HD_RESULT start_unit(pb_video_t *p_pb_video)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videodec_start_list(&(p_pb_video->dec_path), 1)) != HD_OK) {
      printf ("err1\n");
      return ret;
   }

   if ((ret = hd_videoproc_start_list(&(p_pb_video->proc_path), 1)) != HD_OK) {
      printf ("err2\n");
      return ret;
   }

   if ((ret = hd_videoout_start_list(&(p_pb_video->out_path), 1)) != HD_OK) {
      printf ("err3\n");
      return ret;
   }
   return ret;
}

HD_RESULT stop_unit(pb_video_t *p_pb_video)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videodec_stop_list(&(p_pb_video->dec_path), 1)) != HD_OK) {
      return ret;
   }

   if ((ret = hd_videoproc_stop_list(&(p_pb_video->proc_path), 1)) != HD_OK) {
      return ret;
   }

   if ((ret = hd_videoout_stop_list(&(p_pb_video->out_path), 1)) != HD_OK) {
      return ret;
   }
   return ret;
}

int main(int argc, char **argv)
{
   pthread_t pb_thread_id;                    // playback thread
   HD_RESULT ret;                            // return value
   int key;                                 // input key for flow control
```

```
pb_speed_demo_t pb_speed_demo = {
   {                                      // video_path
      {0},                               // out_syscaps
      0,                                 // out_ctrl
      0,                                 // out_path
      0,                                 // dec_path
      0,                                 // proc_path
      {0},                               // lcd_syscaps
   },
   {                                      // speed_ctrl
      {                                  // pb_speed
         {"8x", 1, 8},                   // 0
         {"4x", 1, 4},                   // 1
         {"2x", 1, 2},                   // 2
         {"1x", 1, 1},                   // 3 (default)
         {"1/2 x", 2, 1},                // 4
         {"1/4 x", 4, 1},                // 5
         {"1/8 x", 8, 1}                 // 6
      },
      3,                                 // speed_level
      0,                                 // pb_pause
      0,                                 // pb_exit
   },
   {                                      // pattern
      "1920x1080_ref1",                   // pattern file name
      30,                                // pattern frame rate
      1920,                               // pattern image width
      1080,                               // pattern image height
   },
};

pb_speed_demo_t *p_pb_speed_demo = &pb_speed_demo;
pb_speed_ctrl_t *p_speed_ctrl    = &(p_pb_speed_demo->speed_ctrl);
pb_video_t      *p_video_path    = &(p_pb_speed_demo->video_path);
pb_pattern_t    *p_pattern       = &(p_pb_speed_demo->pattern);

if (argc == 5) {
   strncpy(p_pattern->name, argv[1], sizeof(p_pattern->name));
   p_pattern->frame_rate = atoi(argv[2]);
   p_pattern->img_width = atoi(argv[3]);
   p_pattern->img_height = atoi(argv[4]);
}

//open isf flow
ret = hd_common_init(1);
if (ret != HD_OK) {
   printf("common init fail\n");
   return -1;
}

//init all modules
ret = init_module();
if (ret != HD_OK) {
   printf("init fail\n");
   return -1;
}

//open all modules
ret = open_module(p_video_path);
if (ret != HD_OK) {
   printf("open fail\n");
   return -1;
}

//get video capability
ret = get_video_syscaps(p_video_path);
if (ret != HD_OK) {
   printf("get video syscaps fail\n");
   return -1;
}
```

```
    //set video parameter
    ret = set_video_cfg(p_pb_speed_demo);
    if (ret != HD_OK) {
       printf("set video cfg fail\n");
       return -1;
    }

    //bind video units
    ret = bind_unit();
    if (ret != HD_OK) {
       printf("bind unit fail\n");
       return -1;
    }

    //start video units
    ret = start_unit(p_video_path);
    if (ret != HD_OK) {
       printf("start unit fail\n");
       return -1;
    }

    //create playback_thread
    if (pthread_create(&pb_thread_id, NULL, playback_thread, (void *)p_pb_speed_demo) < 0) {
       printf("create thread playback_thread failed!");
       return -1;
    }

    while (1) {
       printf("Enter q to exit, speed control -> f: faster, s: slower, n: normal, p: pause.\n");
       key = getchar();
       if (key == 'q') {                    // 'q': exit sample program
          p_speed_ctrl->pb_exit = 1;
          break;
       } else if (key == 'f') {             // 'f': speed up playback (up to 8x)
          if (p_speed_ctrl->speed_level > 0) {
             p_speed_ctrl->speed_level--;
          }
          printf("  Speed: %s\n", p_speed_ctrl->pb_speed[p_speed_ctrl->speed_level].speed_str);
       } else if (key == 's') {             // 's': slow down playback (down to 1/8x)
          if (p_speed_ctrl->speed_level < ((sizeof(p_speed_ctrl->pb_speed) / sizeof(pb_speed_t)) - 1)) {
             p_speed_ctrl->speed_level++;
          }
          printf("  Speed: %s \n", p_speed_ctrl->pb_speed[p_speed_ctrl->speed_level].speed_str);
       } else if (key == 'n') {             // 'n': normal playback speed (set to 1x)
          p_speed_ctrl->speed_level = 3;
          printf("  Speed: %s\n", p_speed_ctrl->pb_speed[p_speed_ctrl->speed_level].speed_str);
       } else if (key == 'p') {             // 'p': pause or resume playback
          p_speed_ctrl->pb_pause = (!p_speed_ctrl->pb_pause);
          if (p_speed_ctrl->pb_pause) {
             printf("  Speed: pause!\n");
          } else {
             printf("  Speed: continue!\n");
          }
       }
    }

    //destroy playback thread
    pthread_join(pb_thread_id, NULL);

    //stop video units
    ret = stop_unit(p_video_path);
    if (ret != HD_OK) {
       printf("stop unit fail\n");
       return -1;
    }

    //unbind video units
    ret = unbind_unit();
    if (ret != HD_OK) {
       printf("unbind unit fail\n");
       return -1;
    }
```

```
   //close all modules
   ret = close_module(p_video_path);
   if (ret != HD_OK) {
      printf("close fail\n");
      return -1;
   }

   //uninit all modules
   ret = uninit_module();
   if (ret != HD_OK) {
      printf("uninit fail\n");
      return -1;
   }

   //close isf flow
    ret = hd_common_uninit();
   if (ret != HD_OK) {
      printf("common un init fail\n");
      return -1;
   }

   return 0;
}
```

This example code flow description is shown as the following:

1. Initialize the modules used in Playback: hd_videodec, hd_videoprocess and hd_videoout.
2. Create the decode path, the video process path and the output path.
3. Get the video capability, such as the LCD resolution.
4. Set the parameters of the decode path, such as the video frame rate, the frame width and the frame height. And set the parameters of the output path, such as the output position (x, y) and the dimension (width, height).
5. Connect the paths, the stream flow is built by the following connections:
   hd_videodec_bind(HD_VIDEODEC_OUT(0, 0), HD_VIDEOPROC_IN(0, 0));
   The outport #0 of the decode path connectes to the inport #0 of the video process path.
   hd_videoproc_bind(HD_VIDEOPROC_OUT(0, 0), HD_VIDEOOUT_IN(0, 0));
   The outport #0 of the video process path connectes to the inport #0 of the output path.
6. The stream flow starts running after changing the status of these paths, including the decode path, the video process path and the output path , to “Start”.
7. Create a thread to open the input h.264 file, and then use *hd_videodec_send_list()* API to push the h.264 stream into the decode path.
8. Check the key button in the console window. In this sample code, we use “f” key to speed up the playback, and “s” key to slow down the playback.
9. Press “q” key to close this sample code.
10. Change the status of the decode path, the video process path and the output path to “Stop”.
11. Disconnect all connections.
12. Close all instance paths.
13. Uninitialize all modules.

4.4. Video recod connection

The following figure shows the five stages in this video record example code. Int the first stage, the APIs *hd_xxx_open()* is used to open the instance path. The *hd_xxx_bind()* is to connect each instance path together in the second stage. After changing the status of each instance path to "start" by *hd_xxx_start()*, sensor data will be transmitted and encoded. User could call *hd_videoenc_pull_out_buf()* to get encoded bitstream and save to file. After this bitstream is invalid, user should call *hd_videoenc_release_out_buf()* to release buffer.

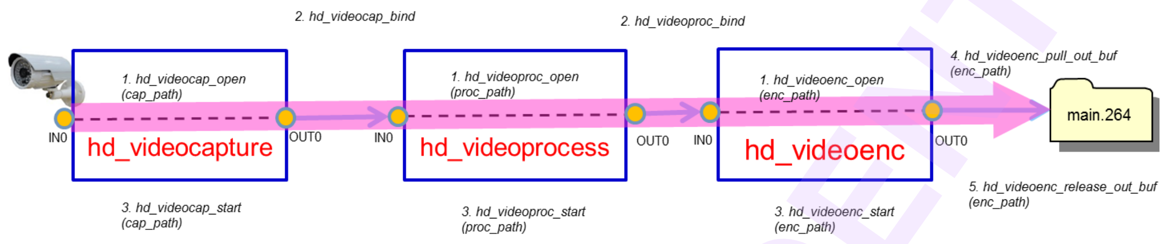
2. hd_videocap_bind
2. hd_videoproc_bind
1. hd_videocap_open (cap_path)
1. hd_videoproc_open (proc_path)
1. hd_videoenc_open (enc_path)
4. hd_videoenc_pull_out_buf (enc_path)
IN0
OUT0
IN0
OUT0
IN0
OUT0
hd_videocapture
hd_videoprocess
hd_videoenc
main.264
3. hd_videocap_start (cap_path)
3. hd_videoproc_start (proc_path)
3. hd_videoenc_start (enc_path)
5. hd_videoenc_release_out_buf (enc_path)

Fig. 9 HDAL Liveview Connect Diagram

Following is the source code of this video record example:

```
/**
	@brief Sample code of video record.\n

	@file video_record.c

	@author Boyan Huang

	@ingroup mhdal

	@note Nothing.

	Copyright Novatek Microelectronics Corp. 2018.  All rights reserved.
*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>
#include "hdal.h"
#include "hd_debug.h"


#define DEBUG_MENU 		1

#define CHKPNT			printf("\033[37mCHK: %s, %s: %d\033[0m\r\n",__FILE__,__func__,__LINE__)
#define DBGH(x)			printf("\033[0;35m%s=0x%08X\033[0m\r\n", #x, x)
#define DBGD(x)			printf("\033[0;35m%s=%d\033[0m\r\n", #x, x)

///////////////////////////////////////////////////////////////////////////////

#define RAW_COMPRESS_RATIO 70

#define DBGINFO_BUFSIZE()	(0x200)

//RAW
#define VDO_RAW_BUFSIZE(w, h, pxlfmt)   (ALIGN_CEIL_4((w) * HD_VIDEO_PXLFMT_BPP(pxlfmt) / 8) * (h))
//RAW compress only support 12bit mode
#define VDO_NRX_BUFSIZE(w, h)           ALIGN_CEIL_4(ALIGN_CEIL_4(ALIGN_CEIL_4((w) * 12 / 8) *
RAW_COMPRESS_RATIO / 100) * (h) + (ALIGN_CEIL_32(w) / 32) * 16 / 8)
//CA for AWB
#define VDO_CA_BUF_SIZE(win_num_w, win_num_h) ((win_num_w * win_num_h << 3) << 1)
//LA for AE
#define VDO_LA_BUF_SIZE(win_num_w, win_num_h) ((win_num_w * win_num_h << 1) << 1)
//YOUT for SHDR/WDR
```

```
#define VDO_YOUT_BUF_SIZE(win_num_w, win_num_h) (ALIGN_CEIL_4(win_num_w * 12 / 8) * win_num_h)
//ETH
#define VDO_ETH_BUF_SIZE(roi_w, roi_h, b_out_sel, b_8bit_sel) ((((roi_w >> (b_out_sel ? 1 : 0)) >>
(b_8bit_sel ? 0 : 2)) * (roi_h - 4)) >> (b_out_sel ? 1 : 0))
//VA for AF
#define VDO_VA_BUF_SIZE(win_num_w, win_num_h) ((win_num_w * win_num_h << 1) << 2)

#define VDO_YUV_BUFSIZE(w, h, pxlfmt)   ALIGN_CEIL_4(((w) * (h) * HD_VIDEO_PXLFMT_BPP(pxlfmt)) / 8)
#define VDO_NVX_BUFSIZE(w, h, pxlfmt)   (VDO_YUV_BUFSIZE(w, h, pxlfmt) * RAW_COMPRESS_RATIO / 100)

#define SEN_OUT_FMT         HD_VIDEO_PXLFMT_RAW12
#define CAP_OUT_FMT         HD_VIDEO_PXLFMT_RAW12
#define CA_WIN_NUM_W        32
#define CA_WIN_NUM_H        32
#define LA_WIN_NUM_W        32
#define LA_WIN_NUM_H        32
#define VA_WIN_NUM_W        16
#define VA_WIN_NUM_H        16
#define YOUT_WIN_NUM_W 128
#define YOUT_WIN_NUM_H 128
#define ETH_8BIT_SEL        0 //0: 2bit out, 1:8 bit out
#define ETH_OUT_SEL         1 //0: full, 1: subsample 1/2

#define VDO_SIZE_W          1920
#define VDO_SIZE_H          1080

///////////////////////////////////////////////////////////////////////////////


HD_RESULT mem_init(void)
{
      HD_RESULT             ret;
      HD_COMMON_MEM_INIT_CONFIG mem_cfg = {0};

      // config common pool (cap)
      mem_cfg.pool_info[0].type = HD_COMMON_MEM_COMMON_POOL;
      mem_cfg.pool_info[0].blk_size = DBGINFO_BUFSIZE()+VDO_RAW_BUFSIZE(VDO_SIZE_W, VDO_SIZE_H,
CAP_OUT_FMT)

      +VDO_CA_BUF_SIZE(CA_WIN_NUM_W, CA_WIN_NUM_H)

      +VDO_LA_BUF_SIZE(LA_WIN_NUM_W, LA_WIN_NUM_H)

      +VDO_YOUT_BUF_SIZE(YOUT_WIN_NUM_W, YOUT_WIN_NUM_H)

      +VDO_ETH_BUF_SIZE(VDO_SIZE_W, VDO_SIZE_H, ETH_OUT_SEL, ETH_8BIT_SEL);
      mem_cfg.pool_info[0].blk_cnt = 3;
      mem_cfg.pool_info[0].ddr_id = DDR_ID0;
      // config common pool (main)
      mem_cfg.pool_info[1].type = HD_COMMON_MEM_COMMON_POOL;
      mem_cfg.pool_info[1].blk_size = DBGINFO_BUFSIZE()+VDO_YUV_BUFSIZE(VDO_SIZE_W, VDO_SIZE_H,
HD_VIDEO_PXLFMT_YUV420);
      mem_cfg.pool_info[1].blk_cnt = 3;
      mem_cfg.pool_info[1].ddr_id = DDR_ID0;

      ret = hd_common_mem_init(&mem_cfg);
      return ret;
}

HD_RESULT mem_exit(void)
{
      HD_RESULT ret = HD_OK;
      hd_common_mem_uninit();
      return ret;
}

///////////////////////////////////////////////////////////////////////////////

HD_RESULT get_cap_caps(HD_PATH_ID video_cap_ctrl, HD_VIDEOCAP_SYSCAPS *p_video_cap_syscaps)
{
      HD_RESULT ret = HD_OK;
      hd_videocap_get(video_cap_ctrl, HD_VIDEOCAP_PARAM_SYSCAPS, p_video_cap_syscaps);
      return ret;
}

HD_RESULT get_cap_sysinfo(HD_PATH_ID video_cap_ctrl)
{
      HD_RESULT ret = HD_OK;
      HD_VIDEOCAP_SYSINFO sys_info = {0};

      hd_videocap_get(video_cap_ctrl, HD_VIDEOCAP_PARAM_SYSINFO, &sys_info);
      printf("sys_info.devid =0x%X, cur_fps[0]=%d/%d, vd_count=%llu\r\n", sys_info.dev_id,
GET_HI_UINT16(sys_info.cur_fps[0]), GET_LO_UINT16(sys_info.cur_fps[0]), sys_info.vd_count);
      return ret;
}

HD_RESULT set_cap_cfg(HD_PATH_ID *p_video_cap_ctrl)
{
      HD_RESULT ret = HD_OK;
      HD_VIDEOCAP_DRV_CONFIG cap_cfg = {0};
      HD_PATH_ID video_cap_ctrl = 0;
      HD_VIDEOCAP_CTRL iq_ctl = {0};

      snprintf(cap_cfg.sen_cfg.sen_dev.driver_name, HD_VIDEOCAP_SEN_NAME_LEN-1, "nvt_sen_imx291");
      cap_cfg.sen_cfg.sen_dev.if_type = HD_COMMON_VIDEO_IN_MIPI_CSI;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.pinmux.sensor_pinmux =  0x220; //PIN_SENSOR_CFG_MIPI |
PIN_SENSOR_CFG_MCLK
      cap_cfg.sen_cfg.sen_dev.pin_cfg.pinmux.serial_if_pinmux = 0xf04;//PIN_MIPI_LVDS_CFG_CLK2 |
PIN_MIPI_LVDS_CFG_DAT0|PIN_MIPI_LVDS_CFG_DAT1 | PIN_MIPI_LVDS_CFG_DAT2 | PIN_MIPI_LVDS_CFG_DAT3
      cap_cfg.sen_cfg.sen_dev.pin_cfg.pinmux.cmd_if_pinmux = 0x10;//PIN_I2C_CFG_CH2
      cap_cfg.sen_cfg.sen_dev.pin_cfg.clk_lane_sel = HD_VIDEOCAP_SEN_CLANE_SEL_CSI0_USE_C2;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[0] = 0;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[1] = 1;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[2] = HD_VIDEOCAP_SEN_IGNORE;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[3] = HD_VIDEOCAP_SEN_IGNORE;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[4] = HD_VIDEOCAP_SEN_IGNORE;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[5] = HD_VIDEOCAP_SEN_IGNORE;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[6] = HD_VIDEOCAP_SEN_IGNORE;
      cap_cfg.sen_cfg.sen_dev.pin_cfg.sen_2_serial_pin_map[7] = HD_VIDEOCAP_SEN_IGNORE;
      ret = hd_videocap_open(0, HD_VIDEOCAP_0_CTRL, &video_cap_ctrl); //open this for device control
      if (ret != HD_OK) {
            return ret;
      }
      ret |= hd_videocap_set(video_cap_ctrl, HD_VIDEOCAP_PARAM_DRV_CONFIG, &cap_cfg);
      iq_ctl.func = HD_VIDEOCAP_FUNC_AE | HD_VIDEOCAP_FUNC_AWB;
      ret |= hd_videocap_set(video_cap_ctrl, HD_VIDEOCAP_PARAM_CTRL, &iq_ctl);

      *p_video_cap_ctrl = video_cap_ctrl;
      return ret;
}

HD_RESULT set_cap_param(HD_PATH_ID video_cap_path, HD_DIM *p_dim)
{
      HD_RESULT ret = HD_OK;
      {//select sensor mode, manually or automatically
            HD_VIDEOCAP_IN video_in_param = {0};

            video_in_param.sen_mode = HD_VIDEOCAP_SEN_MODE_AUTO; //auto select sensor mode by the parameter
of HD_VIDEOCAP_PARAM_OUT
            video_in_param.frc = HD_VIDEO_FRC_RATIO(30,1);
            video_in_param.dim.w = p_dim->w;
            video_in_param.dim.h = p_dim->h;
            video_in_param.pxlfmt = SEN_OUT_FMT;
            video_in_param.out_frame_num = HD_VIDEOCAP_SEN_FRAME_NUM_1;
            ret = hd_videocap_set(video_cap_path, HD_VIDEOCAP_PARAM_IN, &video_in_param);
            //printf("set_cap_param MODE=%d\r\n", ret);
            if (ret != HD_OK) {
                  return ret;
            }
      }
      #if 1 //no crop, full frame
      {
            HD_VIDEOCAP_CROP video_crop_param = {0};
            video_crop_param.mode = HD_CROP_OFF;
            ret = hd_videocap_set(video_cap_path, HD_VIDEOCAP_PARAM_OUT_CROP, &video_crop_param);
            //printf("set_cap_param CROP NONE=%d\r\n", ret);
      }
      #else //HD_CROP_ON
      {
            HD_VIDEOCAP_CROP video_crop_param = {0};

            video_crop_param.mode = HD_CROP_ON;
            video_crop_param.win.rect.x = 0;
            video_crop_param.win.rect.y = 0;
            video_crop_param.win.rect.w = 1920/2;
            video_crop_param.win.rect.h= 1080/2;
            video_crop_param.align.w = 4;
            video_crop_param.align.h = 4;
            ret = hd_videocap_set(video_cap_path, HD_VIDEOCAP_PARAM_OUT_CROP, &video_crop_param);
            //printf("set_cap_param CROP ON=%d\r\n", ret);
      }
      #endif
      {
            HD_VIDEOCAP_OUT video_out_param = {0};

            //without setting dim for no scaling, using original sensor out size
            video_out_param.pxlfmt = CAP_OUT_FMT;
            video_out_param.dir = HD_VIDEO_DIR_NONE;
            ret = hd_videocap_set(video_cap_path, HD_VIDEOCAP_PARAM_OUT, &video_out_param);
            //printf("set_cap_param OUT=%d\r\n", ret);
      }

      return ret;
}

///////////////////////////////////////////////////////////////////////////////

HD_RESULT set_proc_cfg(HD_PATH_ID *p_video_proc_ctrl, HD_DIM* p_max_dim)
{
      HD_RESULT ret = HD_OK;
      HD_VIDEOPROC_DEV_CONFIG video_cfg_param = {0};
      HD_VIDEOPROC_CTRL video_ctrl_param = {0};
      HD_PATH_ID video_proc_ctrl = 0;

      ret = hd_videoproc_open(0, HD_VIDEOPROC_0_CTRL, &video_proc_ctrl); //open this for device control
      if (ret != HD_OK)
            return ret;

      if (p_max_dim != NULL ) {
            video_cfg_param.pipe = HD_VIDEOPROC_PIPE_RAWALL;
            video_cfg_param.isp_id = 0;
            video_cfg_param.ctrl_max.func = 0;
            video_cfg_param.in_max.func = 0;
            video_cfg_param.in_max.dim.w = p_max_dim->w;
            video_cfg_param.in_max.dim.h = p_max_dim->h;
            video_cfg_param.in_max.pxlfmt = CAP_OUT_FMT;
            video_cfg_param.in_max.frc = HD_VIDEO_FRC_RATIO(1,1);
            ret = hd_videoproc_set(video_proc_ctrl, HD_VIDEOPROC_PARAM_DEV_CONFIG, &video_cfg_param);
            if (ret != HD_OK) {
                  return HD_ERR_NG;
            }
      }

      video_ctrl_param.func = 0;
      ret = hd_videoproc_set(video_proc_ctrl, HD_VIDEOPROC_PARAM_CTRL, &video_ctrl_param);

      *p_video_proc_ctrl = video_proc_ctrl;

      return ret;
}

HD_RESULT set_proc_param(HD_PATH_ID video_proc_path, HD_DIM* p_dim)
{
      HD_RESULT ret = HD_OK;
```

```
	if (p_dim != NULL) { //if videoproc is already binding to dest module, not require to setting this!
		HD_VIDEOPROC_OUT video_out_param = {0};
		video_out_param.func = 0;
		video_out_param.dim.w = p_dim->w;
		video_out_param.dim.h = p_dim->h;
		video_out_param.pxlfmt = HD_VIDEO_PXLFMT_YUV420;
		video_out_param.dir = HD_VIDEO_DIR_NONE;
		video_out_param.frc = HD_VIDEO_FRC_RATIO(1,1);
		ret = hd_videoproc_set(video_proc_path, HD_VIDEOPROC_PARAM_OUT, &video_out_param);
	}

	return ret;
}

///////////////////////////////////////////////////////////////////////////////

HD_RESULT set_enc_cfg(HD_PATH_ID video_enc_path, HD_DIM *p_max_dim, UINT32 enc_type, UINT32 max_bitrate)
{
	HD_RESULT ret = HD_OK;
	HD_VIDEOENC_PATH_CONFIG video_path_config = {0};

	if (p_max_dim != NULL) {

		//--- HD_VIDEOENC_PARAM_PATH_CONFIG ---
		if (enc_type == 0) {
		video_path_config.max_mem.codec_type = HD_CODEC_TYPE_H265;
		} else if (enc_type == 1) {
		video_path_config.max_mem.codec_type = HD_CODEC_TYPE_H264;
		} else {
		video_path_config.max_mem.codec_type = HD_CODEC_TYPE_JPEG;
		}
		video_path_config.max_mem.max_dim.w  = p_max_dim->w;
		video_path_config.max_mem.max_dim.h  = p_max_dim->h;
		video_path_config.max_mem.bitrate    = max_bitrate;
		video_path_config.max_mem.enc_buf_ms = 3000;
		video_path_config.max_mem.svc_layer  = HD_SVC_4X;
		video_path_config.max_mem.ltr        = TRUE;
		video_path_config.max_mem.rotate     = FALSE;
		video_path_config.max_mem.source_output   = FALSE;
		video_path_config.isp_id            = 0;
		ret = hd_videoenc_set(video_enc_path, HD_VIDEOENC_PARAM_PATH_CONFIG, &video_path_config);
		if (ret != HD_OK) {
			printf("set_enc_path_config = %d\r\n", ret);
			return HD_ERR_NG;
		}
	}

	return ret;
}

HD_RESULT set_enc_param(HD_PATH_ID video_enc_path, HD_DIM *p_dim, UINT32 enc_type, UINT32 bitrate)
{
	HD_RESULT ret = HD_OK;
	HD_VIDEOENC_IN  video_in_param = {0};
	HD_VIDEOENC_OUT video_out_param = {0};
	HD_H26XENC_RATE_CONTROL rc_param = {0};

	if (p_dim != NULL) {

		//--- HD_VIDEOENC_PARAM_IN ---
		video_in_param.dir           = HD_VIDEO_DIR_NONE;
		video_in_param.pxl_fmt = HD_VIDEO_PXLFMT_YUV420;
		video_in_param.dim.w   = p_dim->w;
		video_in_param.dim.h   = p_dim->h;
		ret = hd_videoenc_set(video_enc_path, HD_VIDEOENC_PARAM_IN, &video_in_param);
		if (ret != HD_OK) {
			printf("set_enc_param_in = %d\r\n", ret);
			return ret;
		}

		printf("enc_type=%d\r\n", enc_type);
```

```
            if (enc_type == 0) {

                  //--- HD_VIDEOENC_PARAM_OUT_ENC_PARAM ---
                  video_out_param.codec_type         = HD_CODEC_TYPE_H265;
                  video_out_param.h26x.profile       = HD_H265E_MAIN_PROFILE;
                  video_out_param.h26x.level_idc     = HD_H265E_LEVEL_5;
                  video_out_param.h26x.gop_num       = 15;
                  video_out_param.h26x.ltr_interval  = 0;
                  video_out_param.h26x.ltr_pre_ref   = 0;
                  video_out_param.h26x.gray_en       = 0;
                  video_out_param.h26x.source_output = 0;
                  video_out_param.h26x.svc_layer     = HD_SVC_DISABLE;
                  video_out_param.h26x.entropy_mode  = HD_H265E_CABAC_CODING;
                  ret = hd_videoenc_set(video_enc_path, HD_VIDEOENC_PARAM_OUT_ENC_PARAM,
&video_out_param);
                  if (ret != HD_OK) {
                        printf("set_enc_param_out = %d\r\n", ret);
                        return ret;
                  }

                  //--- HD_VIDEOENC_PARAM_OUT_RATE_CONTROL ---
                  rc_param.rc_mode             = HD_RC_MODE_CBR;
                  rc_param.cbr.bitrate         = bitrate;
                  rc_param.cbr.frame_rate_base = 30;
                  rc_param.cbr.frame_rate_incr = 1;
                  rc_param.cbr.init_i_qp       = 26;
                  rc_param.cbr.min_i_qp        = 10;
                  rc_param.cbr.max_i_qp        = 45;
                  rc_param.cbr.init_p_qp       = 26;
                  rc_param.cbr.min_p_qp        = 10;
                  rc_param.cbr.max_p_qp        = 45;
                  rc_param.cbr.static_time     = 4;
                  rc_param.cbr.ip_weight       = 0;
                  ret = hd_videoenc_set(video_enc_path, HD_VIDEOENC_PARAM_OUT_RATE_CONTROL, &rc_param);
                  if (ret != HD_OK) {
                        printf("set_enc_rate_control = %d\r\n", ret);
                        return ret;
                  }
            } else if (enc_type == 1) {

                  //--- HD_VIDEOENC_PARAM_OUT_ENC_PARAM ---
                  video_out_param.codec_type         = HD_CODEC_TYPE_H264;
                  video_out_param.h26x.profile       = HD_H264E_HIGH_PROFILE;
                  video_out_param.h26x.level_idc     = HD_H264E_LEVEL_5_1;
                  video_out_param.h26x.gop_num       = 15;
                  video_out_param.h26x.ltr_interval  = 0;
                  video_out_param.h26x.ltr_pre_ref   = 0;
                  video_out_param.h26x.gray_en       = 0;
                  video_out_param.h26x.source_output = 0;
                  video_out_param.h26x.svc_layer     = HD_SVC_DISABLE;
                  video_out_param.h26x.entropy_mode  = HD_H264E_CABAC_CODING;
                  ret = hd_videoenc_set(video_enc_path, HD_VIDEOENC_PARAM_OUT_ENC_PARAM,
&video_out_param);
                  if (ret != HD_OK) {
                        printf("set_enc_param_out = %d\r\n", ret);
                        return ret;
                  }

                  //--- HD_VIDEOENC_PARAM_OUT_RATE_CONTROL ---
                  rc_param.rc_mode             = HD_RC_MODE_CBR;
                  rc_param.cbr.bitrate         = bitrate;
                  rc_param.cbr.frame_rate_base = 30;
                  rc_param.cbr.frame_rate_incr = 1;
                  rc_param.cbr.init_i_qp       = 26;
                  rc_param.cbr.min_i_qp        = 10;
                  rc_param.cbr.max_i_qp        = 45;
                  rc_param.cbr.init_p_qp       = 26;
                  rc_param.cbr.min_p_qp        = 10;
                  rc_param.cbr.max_p_qp        = 45;
                  rc_param.cbr.static_time     = 4;
                  rc_param.cbr.ip_weight       = 0;
                  ret = hd_videoenc_set(video_enc_path, HD_VIDEOENC_PARAM_OUT_RATE_CONTROL, &rc_param);
                  if (ret != HD_OK) {
                        printf("set_enc_rate_control = %d\r\n", ret);
                        return ret;
                  }

            } else if (enc_type == 2) {

                  //--- HD_VIDEOENC_PARAM_OUT_ENC_PARAM ---
                  video_out_param.codec_type        = HD_CODEC_TYPE_JPEG;
                  video_out_param.jpeg.retstart_interval = 0;
                  video_out_param.jpeg.image_quality = 70;
                  ret = hd_videoenc_set(video_enc_path, HD_VIDEOENC_PARAM_OUT_ENC_PARAM,
&video_out_param);
                  if (ret != HD_OK) {
                        printf("set_enc_param_out = %d\r\n", ret);
                        return ret;
                  }

            } else {

                  printf("not support enc_type\r\n");
                  return HD_ERR_NG;
            }
      }

      return ret;
}

////////////////////////////////////////////////////////////////////////////////

typedef struct _VIDEO_RECORD {

      // (1)
      HD_VIDEOCAP_SYSCAPS cap_syscaps;
      HD_PATH_ID cap_ctrl;
      HD_PATH_ID cap_path;

      HD_DIM  cap_dim;
      HD_DIM  proc_max_dim;

      // (2)
      HD_VIDEOPROC_SYSCAPS proc_syscaps;
      HD_PATH_ID proc_ctrl;
      HD_PATH_ID proc_path;

      HD_DIM  enc_max_dim;
      HD_DIM  enc_dim;

      // (3)
      HD_VIDEOENC_SYSCAPS enc_syscaps;
      HD_PATH_ID enc_path;

      // (4) user pull
      pthread_t  enc_thread_id;
      UINT32     enc_exit;
      UINT32     flow_start;

} VIDEO_RECORD;

HD_RESULT init_module(void)
{
      HD_RESULT ret;
      if ((ret = hd_videocap_init()) != HD_OK)
            return ret;
      if ((ret = hd_videoproc_init()) != HD_OK)
            return ret;
    if ((ret = hd_videoenc_init()) != HD_OK)
            return ret;
      return HD_OK;
}

HD_RESULT open_module(VIDEO_RECORD *p_stream, HD_DIM* p_proc_max_dim)
```

```
{
      HD_RESULT ret;
      // set videocap config
      ret = set_cap_cfg(&p_stream->cap_ctrl);
      if (ret != HD_OK) {
            printf("set cap-cfg fail=%d\n", ret);
            return HD_ERR_NG;
      }
      // set videoproc config
      ret = set_proc_cfg(&p_stream->proc_ctrl, p_proc_max_dim);
      if (ret != HD_OK) {
            printf("set proc-cfg fail=%d\n", ret);
            return HD_ERR_NG;
      }

      if ((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_0, &p_stream->cap_path)) != HD_OK)
            return ret;
      if ((ret = hd_videoproc_open(HD_VIDEOPROC_0_IN_0, HD_VIDEOPROC_0_OUT_0, &p_stream->proc_path)) !=
HD_OK)
            return ret;
      if ((ret = hd_videoenc_open(HD_VIDEOENC_0_IN_0, HD_VIDEOENC_0_OUT_0, &p_stream->enc_path)) != HD_OK)
            return ret;

      return HD_OK;
}

HD_RESULT close_module(VIDEO_RECORD *p_stream)
{
      HD_RESULT ret;
      if ((ret = hd_videocap_close(p_stream->cap_path)) != HD_OK)
            return ret;
      if ((ret = hd_videoproc_close(p_stream->proc_path)) != HD_OK)
            return ret;
      if ((ret = hd_videoenc_close(p_stream->enc_path)) != HD_OK)
            return ret;
      return HD_OK;
}

HD_RESULT exit_module(void)
{
      HD_RESULT ret;
      if ((ret = hd_videocap_uninit()) != HD_OK)
            return ret;
      if ((ret = hd_videoproc_uninit()) != HD_OK)
            return ret;
      if ((ret = hd_videoenc_uninit()) != HD_OK)
            return ret;
      return HD_OK;
}

static void *encode_thread(void *arg)
{
      VIDEO_RECORD* p_stream0 = (VIDEO_RECORD *)arg;
      HD_RESULT ret = HD_OK;
      HD_VIDEOENC_BS  data_pull;
      UINT32 j;

      UINT32 vir_addr_main;
      HD_VIDEOENC_BUFINFO phy_buf_main;
      char file_path_main[32] = "/mnt/sd/dump_bs_main.dat";
      FILE *f_out_main;
      #define PHY2VIRT_MAIN(pa) (vir_addr_main + (pa - phy_buf_main.buf_info.phy_addr))

      //------ wait flow_start ------
      while (p_stream0->flow_start == 0) sleep(1);

      // query physical address of bs buffer ( this can ONLY query after hd_videoenc_start() is called !! )
      hd_videoenc_get(p_stream0->enc_path, HD_VIDEOENC_PARAM_BUFINFO, &phy_buf_main);

      // mmap for bs buffer (just mmap one time only, calculate offset to virtual address later)
      vir_addr_main = (UINT32)hd_common_mem_mmap(HD_COMMON_MEM_MEM_TYPE_CACHE,
phy_buf_main.buf_info.phy_addr, phy_buf_main.buf_info.buf_size);
```

```
    //----- open output files -----
    if ((f_out_main = fopen(file_path_main, "wb")) == NULL) {
        HD_VIDEOENC_ERR("open file (%s) fail....\r\n", file_path_main);
    } else {
        printf("\r\ndump main bitstream to file (%s) ....\r\n", file_path_main);
    }

    printf("\r\nif you want to stop, enter \"q\" to exit !!\r\n\r\n");

    //--------- pull data test ---------
    while (p_stream0->enc_exit == 0) {
        //pull data
        ret = hd_videoenc_pull_out_buf(p_stream0->enc_path, &data_pull, -1);

        if (ret == HD_OK) {
            for (j=0; j< data_pull.pack_num; j++) {
                UINT8 *ptr = (UINT8 *)PHY2VIRT_MAIN(data_pull.video_pack[j].phy_addr);
                UINT32 len = data_pull.video_pack[j].size;
                if (f_out_main) fwrite(ptr, 1, len, f_out_main);
                if (f_out_main) fflush(f_out_main);
            }

            // release data
            ret = hd_videoenc_release_out_buf(p_stream0->enc_path, &data_pull);
        }
    }

    // mummap for bs buffer
    hd_common_mem_munmap((void *)vir_addr_main, phy_buf_main.buf_info.buf_size);

    // close output file
    if (f_out_main) fclose(f_out_main);

    return 0;
}

int main(int argc, char** argv)
{
    HD_RESULT ret;
    INT key;
    VIDEO_RECORD stream[1] = {0}; //0: main stream
    UINT32 enc_type = 0;

    // query program options
    if (argc == 2) {
        enc_type = atoi(argv[1]);
        printf("enc_type %d\r\n", enc_type);
        if(enc_type > 2) {
            printf("error: not support enc_type!\r\n");
            return 0;
        }
    }

    // init hdal
    ret = hd_common_init(0);
    if (ret != HD_OK) {
        printf("common fail=%d\n", ret);
        goto exit;
    }

    // init memory
    ret = mem_init();
    if (ret != HD_OK) {
        printf("mem fail=%d\n", ret);
        goto exit;
    }

    // init all modules
    ret = init_module();
    if (ret != HD_OK) {
        printf("init fail=%d\n", ret);
```

```
            goto exit;
        }

        // open video_record modules (main)
        stream[0].proc_max_dim.w = VDO_SIZE_W; //assign by user
        stream[0].proc_max_dim.h = VDO_SIZE_H; //assign by user
        ret = open_module(&stream[0], &stream[0].proc_max_dim);
        if (ret != HD_OK) {
            printf("open fail=%d\n", ret);
            goto exit;
        }

        // get videocap capability
        ret = get_cap_caps(stream[0].cap_ctrl, &stream[0].cap_syscaps);
        if (ret != HD_OK) {
            printf("get cap-caps fail=%d\n", ret);
            goto exit;
        }

        // set videocap parameter
        stream[0].cap_dim.w = VDO_SIZE_W; //assign by user
        stream[0].cap_dim.h = VDO_SIZE_H; //assign by user
        ret = set_cap_param(stream[0].cap_path, &stream[0].cap_dim);
        if (ret != HD_OK) {
            printf("set cap fail=%d\n", ret);
            goto exit;
        }

        // set videoproc parameter (main)
        ret = set_proc_param(stream[0].proc_path, NULL);
        if (ret != HD_OK) {
            printf("set proc fail=%d\n", ret);
            goto exit;
        }

        // set videoenc config (main)
        stream[0].enc_max_dim.w = VDO_SIZE_W; //assign by user
        stream[0].enc_max_dim.h = VDO_SIZE_H; //assign by user
        ret = set_enc_cfg(stream[0].enc_path, &stream[0].enc_max_dim, enc_type, 2 * 1024 * 1024);
        if (ret != HD_OK) {
            printf("set enc-cfg fail=%d\n", ret);
            return HD_ERR_NG;
        }

        // set videoenc parameter (main)
        stream[0].enc_dim.w = VDO_SIZE_W; //assign by user
        stream[0].enc_dim.h = VDO_SIZE_H; //assign by user
        ret = set_enc_param(stream[0].enc_path, &stream[0].enc_dim, enc_type, 2 * 1024 * 1024);
        if (ret != HD_OK) {
            printf("set enc fail=%d\n", ret);
            goto exit;
        }

        // bind video_record modules (main)
        hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOPROC_0_IN_0);
        hd_videoproc_bind(HD_VIDEOPROC_0_OUT_0, HD_VIDEOENC_0_IN_0);

        // create encode_thread (pull_out bitstream)
        ret = pthread_create(&stream[0].enc_thread_id, NULL, encode_thread, (void *)stream);
        if (ret < 0) {
            printf("create encode thread failed");
            goto exit;
        }

        // start video_record modules (main)
        hd_videocap_start(stream[0].cap_path);
        hd_videoproc_start(stream[0].proc_path);
        // wait ae/awb stable
        sleep(1);
        hd_videoenc_start(stream[0].enc_path);

        // let encode_thread start to work
```

```
	stream[0].flow_start = 1;

	// query user key
	printf("Enter q to exit\n");
	while (1) {
		key = getchar();
		if (key == 'q' || key == 0x3) {
			// let encode_thread stop loop and exit
			stream[0].enc_exit = 1;
			// quit program
			break;
		}

		#if (DEBUG_MENU == 1)
		if (key == 'd') {
			// enter debug menu
			hd_debug_run_menu();
			printf("\r\nEnter q to exit, Enter d to debug\r\n");
		}
		#endif
	}

	// destroy encode thread
	pthread_join(stream[0].enc_thread_id, NULL);

	// stop video_record modules (main)
	hd_videocap_stop(stream[0].cap_path);
	hd_videoproc_stop(stream[0].proc_path);
	hd_videoenc_stop(stream[0].enc_path);

	// unbind video_record modules (main)
	hd_videocap_unbind(HD_VIDEOCAP_0_OUT_0);
	hd_videoproc_unbind(HD_VIDEOPROC_0_OUT_0);

exit:
	// close video_record modules (main)
	ret = close_module(&stream[0]);
	if (ret != HD_OK) {
		printf("close fail=%d\n", ret);
	}

	// uninit all modules
	ret = exit_module();
	if (ret != HD_OK) {
		printf("exit fail=%d\n", ret);
	}

	// uninit memory
	ret = mem_exit();
	if (ret != HD_OK) {
		printf("mem fail=%d\n", ret);
	}

	// uninit hdal
	ret = hd_common_uninit();
	if (ret != HD_OK) {
		printf("common fail=%d\n", ret);
	}

	return 0;
}
```

This flow of the example code is described as the following steps:

1. Initialize the modules used in Videorecord: hd_videocapture, hd_videoprocess and hd_videoenc.
2. Create the capture path, the video process path and the video encode path.
3. Get the video capability, such as the videocap capability.
4. Set the parameters of the capture path, such as the frame width and the frame height. And set the parameters of the video_encode path, such as the dimension (width, height), encode type and bitrate.
5. Connect the paths, the stream flow is built by the following connections:

hd_videodec_bind(HD_VIDEOCAP_OUT(0,0), HD_VIDEOPROC_IN(0,0));
The outport #0 of the capture path connectes to the inport #0 of the video process path.
hd_videoproc_bind(HD_VIDEOPROC_OUT(0,0), HD_VIDEOENC_IN(0,0));
The outport #0 of the video process path connectes to the inport #0 of the video encode path.

6. Create a thread to use *hd_videoenc_pull_out_buf ()* API to get the encoded stream. Then store them into the specified file and use *hd_videoenc_release_out_buf()* API to release data.
7. The stream flow starts running after changing the status of the paths, including the capture path, the video process path and the video encode path, to “Start”.
8. Check the key button in the console window. In this sample code, we use “d” key to enter the debug menu.
9. Press “q” key to close this sample code.
10. Change the status of the capture path, the video process path and the video encode path to “Stop”.
11. Disconnect all connections.
12. Close all instance paths.
13. Uninitialize all modules.

5. Complicated Use Cases

5.1. Overview

It shows examples related to PIP and Sound Selection at PIP status.

5.2. PIP

PIP connection is used by three streaming modules, hd_videocapture, hd_videoprocess and hd_videoout. The hd_videocapture is binded to hd_videoprocess, it receives media from video source, and then pass to hd_videoprocess. The hd_videoprocess is binded to hd_videoout, which can process media, like crop or scale, and then pass to hd_videoout. The hd_videoout will merge multiple layer media, and can output to display device. **Fig. 10** show how these streaming modules are connected for PIP connection.

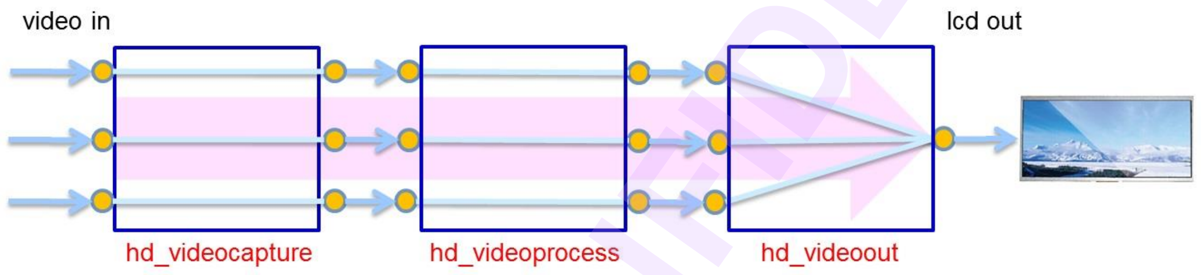

Fig. 10 Streaming modules for PIP connection

Following is the full code of Liveview with PIP example.

```
/**
 * @file liveview_with_pip.c
 * @brief liveview with pip
 * @author Schumy Chen
 * @date in the year 2018
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>
#include "hdal.h"

#define SAMPLE_LV_COUNT   3
#define ALIGN2_DOWN(x)     (((x) >> 1) << 1)

typedef struct _VIDEO_LIVEVIEW {
   // video capture module
   HD_PATH_ID cap_path[SAMPLE_LV_COUNT];
   // video process module
   HD_PATH_ID proc_path[SAMPLE_LV_COUNT];
   //video out module
   HD_PATH_ID out_ctrl;
   HD_PATH_ID out_path[SAMPLE_LV_COUNT];
   HD_VIDEOOUT_SYSCAPS lcd_syscaps;
} VIDEO_LIVEVIEW;

void get_cap_w_h(HD_PATH_ID cap_path_id, UINT32 *w, UINT32 *h)
{
   HD_VIDEOCAP_SYSCAPS cap_info;
   HD_RESULT ret;

   ret = hd_videocap_get(cap_path_id, HD_VIDEOCAP_PARAM_SYSCAPS, &cap_info);
   if (ret != HD_OK) {
      printf("hd_videocap_get:HD_VIDEOCAP_PARAM_SYSCAPS, ret(%x) fail\n", ret);
      exit(0);
   }
   if (w) {
      *w = cap_info.max_dim.w;
   }
   if (h) {
      *h = cap_info.max_dim.h;
   }
}

HD_RESULT get_out_syscaps(HD_PATH_ID video_out_ctrl, HD_VIDEOOUT_SYSCAPS *p_lcd_syscaps)
{
   HD_RESULT ret;

   ret = hd_videoout_get(video_out_ctrl, HD_VIDEOOUT_PARAM_SYSCAPS, p_lcd_syscaps);
   if (ret != HD_OK) {
      printf("hd_videoout_get:param_id(%d) video_out_ctrl(%#lx) fail\n", HD_VIDEOOUT_PARAM_SYSCAPS,
video_out_ctrl);
   }
   return ret;
}

HD_RESULT set_pip_cfg(VIDEO_LIVEVIEW liveview_info)
{
   HD_VIDEOPROC_CROP crop;
   HD_VIDEOPROC_CROP crop_psr;
   HD_VIDEOOUT_WIN_ATTR win;
   HD_VIDEOOUT_WIN_PSR_ATTR win_psr;
   UINT32 cap_w, cap_h;
   HD_RESULT ret = HD_OK;
   int i;
   HD_VIDEOPROC_CROP proc_crop;
   HD_VIDEOPROC_OUT proc_out;
   HD_VIDEOCAP_OUT cap_out;
   HD_VIDEOCAP_PATH_CONFIG cap_config;
   HD_VIDEOPROC_DEV_CONFIG vpe_config;

   for (i = 0; i < SAMPLE_LV_COUNT; i++) {
      /* Set videocap out pool */
      memset(&cap_config, 0x0, sizeof(HD_VIDEOCAP_PATH_CONFIG));
      cap_config.data_pool[0].mode = HD_VIDEOCAP_POOL_ENABLE;
      cap_config.data_pool[0].ddr_id = 0;
      cap_config.data_pool[0].counts = HD_VIDEOCAP_SET_COUNT(3, 0);
      cap_config.data_pool[0].max_counts = HD_VIDEOCAP_SET_COUNT(3, 0);

      cap_config.data_pool[1].mode = HD_VIDEOCAP_POOL_DISABLE;
      cap_config.data_pool[2].mode = HD_VIDEOCAP_POOL_DISABLE;
      cap_config.data_pool[3].mode = HD_VIDEOCAP_POOL_DISABLE;

      ret = hd_videocap_set(liveview_info.cap_path[i], HD_VIDEOCAP_PARAM_PATH_CONFIG, &cap_config);
      if (ret != HD_OK) {
         return ret;
      }

      /* Set videoprocess out pool */
      memset(&vpe_config, 0x0, sizeof(HD_VIDEOPROC_DEV_CONFIG));
      vpe_config.data_pool[0].mode = HD_VIDEOPROC_POOL_ENABLE;
      vpe_config.data_pool[0].ddr_id = 0;
      vpe_config.data_pool[0].counts = HD_VIDEOPROC_SET_COUNT(3, 0);
      vpe_config.data_pool[0].max_counts = HD_VIDEOPROC_SET_COUNT(3, 0);

      vpe_config.data_pool[1].mode = HD_VIDEOPROC_POOL_DISABLE;
      vpe_config.data_pool[2].mode = HD_VIDEOPROC_POOL_DISABLE;
      vpe_config.data_pool[3].mode = HD_VIDEOPROC_POOL_DISABLE;

      ret = hd_videoproc_set(liveview_info.proc_path[i], HD_VIDEOPROC_PARAM_DEV_CONFIG, &vpe_config);
      if (ret != HD_OK) {
         return ret;
      }
```

```
    /* Set videocap output setting */
    cap_out.dim.w = liveview_info.lcd_syscaps.input_dim.w/2;
    cap_out.dim.h = liveview_info.lcd_syscaps.input_dim.h;
    cap_out.pxlfmt = HD_VIDEO_PXLFMT_YUV420_NVX3;//cap_syscaps.pxlfmt;
    ret = hd_videocap_set(liveview_info.cap_path[i], HD_VIDEOCAP_PARAM_OUT, &cap_out);
    if (ret != HD_OK) {
       return ret;
    }

    /* Set videoproc output setting */
    proc_crop.win.rect.x = 0;
    proc_crop.win.rect.y = 0;
    proc_crop.win.rect.w = liveview_info.lcd_syscaps.input_dim.w;
    proc_crop.win.rect.h = liveview_info.lcd_syscaps.input_dim.h;
    proc_crop.win.coord.w = liveview_info.lcd_syscaps.input_dim.w;
    proc_crop.win.coord.h = liveview_info.lcd_syscaps.input_dim.h;
    ret = hd_videoproc_set(liveview_info.proc_path[i], HD_VIDEOPROC_PARAM_OUT_CROP, &proc_crop);
    if (ret != HD_OK) {
       return ret;
    }
    proc_out.pxlfmt = HD_VIDEO_PXLFMT_YUV422_ONE;
    proc_out.dir = HD_VIDEO_DIR_NONE;
    ret = hd_videoproc_set(liveview_info.proc_path[i], HD_VIDEOPROC_PARAM_OUT, &proc_out);
    if (ret != HD_OK) {
       return ret;
    }

    /*    +----------+----------+
     *    |          |          |
     *    |  LV0     |  LV1     |
     *    |          |          |
     *    | +-----+  |  +-----+ |
     *    | | LV0'|  |  | LV2 | |
     *    | +-----+  |  +-----+ |
     *    |          |          |
     *    +----------+----------+
     */

    // LV0: liveview, left, bg_layer
    if(i == 0){
       get_cap_w_h(liveview_info.cap_path[0], &cap_w, &cap_h);
       crop.mode = HD_CROP_ON;
       crop.win.coord.w = liveview_info.lcd_syscaps.input_dim.w;//cap_w;
       crop.win.coord.h = liveview_info.lcd_syscaps.input_dim.h;//cap_h;
       crop.win.rect.x = 0;
       crop.win.rect.y = 0;
       crop.win.rect.w = cap_w / 4;
       crop.win.rect.h = cap_h / 4;
       ret = hd_videoproc_set(liveview_info.proc_path[0], HD_VIDEOPROC_PARAM_OUT_CROP, &crop);
       if (ret != HD_OK) {
          return ret;
       }

       win.rect.x = 0;
       win.rect.y = 0;
       win.rect.w = liveview_info.lcd_syscaps.input_dim.w / 2;
       win.rect.h = liveview_info.lcd_syscaps.input_dim.h;
       win.visible = 1;
       win.layer = HD_LAYER1;
       ret = hd_videoout_set(liveview_info.out_path[0], HD_VIDEOOUT_PARAM_IN_WIN_ATTR, &win);
       if (ret != HD_OK) {
          return ret;
       }

       // LV0': liveview, left, psr-win
       crop_psr.mode = HD_CROP_OFF;
       ret = hd_videoproc_set(liveview_info.proc_path[0], HD_VIDEOPROC_PARAM_OUT_CROP_PSR, &crop_psr);
       if (ret != HD_OK) {
          return ret;
       }

       win_psr.rect.x = liveview_info.lcd_syscaps.input_dim.w / 8;
```

```
        win_psr.rect.y = ALIGN2_DOWN(liveview_info.lcd_syscaps.input_dim.h / 2 -
liveview_info.lcd_syscaps.input_dim.h / 8);
        win_psr.rect.w = liveview_info.lcd_syscaps.input_dim.w / 5;
        win_psr.rect.h = liveview_info.lcd_syscaps.input_dim.h / 5;
        win_psr.visible = 1;
        ret = hd_videoout_set(liveview_info.out_path[0], HD_VIDEOOUT_PARAM_IN_WIN_PSR_ATTR, &win_psr);
        if (ret != HD_OK) {
           return ret;
        }
      }
      // LV1: liveview, right, bg_layer
      else if(i == 1){
        crop.mode = HD_CROP_OFF;
        ret = hd_videoproc_set(liveview_info.proc_path[1], HD_VIDEOPROC_PARAM_OUT_CROP, &crop);
        if (ret != HD_OK) {
           return ret;
        }

        win.rect.x = liveview_info.lcd_syscaps.input_dim.w / 2;
        win.rect.y = 0;
        win.rect.w = liveview_info.lcd_syscaps.input_dim.w / 2;
        win.rect.h = liveview_info.lcd_syscaps.input_dim.h;
        win.visible = 1;
        win.layer = HD_LAYER1;
        ret = hd_videoout_set(liveview_info.out_path[1], HD_VIDEOOUT_PARAM_IN_WIN_ATTR, &win);
        if (ret != HD_OK) {
           return ret;
        }
      }
      // LV2: liveview, right, top-layer
      else if(i == 2){
        get_cap_w_h(liveview_info.cap_path[0], &cap_w, &cap_h);
        crop.mode = HD_CROP_ON;
        crop.win.coord.w = liveview_info.lcd_syscaps.input_dim.w;//cap_w;
        crop.win.coord.h = liveview_info.lcd_syscaps.input_dim.h;//cap_h;
        crop.win.rect.x = 0;
        crop.win.rect.y = 0;
        crop.win.rect.w = cap_w / 4;
        crop.win.rect.h = cap_h / 4;
        ret = hd_videoproc_set(liveview_info.proc_path[2], HD_VIDEOPROC_PARAM_OUT_CROP, &crop);
        if (ret != HD_OK) {
           return ret;
        }

        win.rect.x = liveview_info.lcd_syscaps.input_dim.w / 2 + liveview_info.lcd_syscaps.input_dim.w
/ 8;
        win.rect.y = ALIGN2_DOWN(liveview_info.lcd_syscaps.input_dim.h / 2 -
liveview_info.lcd_syscaps.input_dim.h / 8);
        win.rect.w = liveview_info.lcd_syscaps.input_dim.w / 5;
        win.rect.h = liveview_info.lcd_syscaps.input_dim.h / 5;
        win.visible = 1;
        win.layer = HD_LAYER2;
        ret = hd_videoout_set(liveview_info.out_path[2], HD_VIDEOOUT_PARAM_IN_WIN_ATTR, &win);
        if (ret != HD_OK) {
           return ret;
        }
      }
   }

   return ret;
}

HD_RESULT init_module(void)
{
   HD_RESULT ret = HD_OK;
   if ((ret = hd_videocap_init()) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_init()) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoout_init()) != HD_OK) {
        return ret;
    }

    return ret;
}

HD_RESULT uninit_module(void)
{
    HD_RESULT ret = HD_OK;
    if ((ret = hd_videocap_uninit()) != HD_OK) {
        return ret;
    }
    if ((ret = hd_videoproc_uninit()) != HD_OK) {
        return ret;
    }
    if ((ret = hd_videoout_uninit()) != HD_OK) {
        return ret;
    }

    return ret;
}

HD_RESULT open_module(VIDEO_LIVEVIEW *p_liveview_info)
{
    HD_RESULT ret = HD_OK;
    if ((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_0,
&p_liveview_info->cap_path[0])) != HD_OK) {
        return ret;
    }
    if ((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_1,
&p_liveview_info->cap_path[1])) != HD_OK) {
        return ret;
    }
    if ((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_2,
&p_liveview_info->cap_path[2])) != HD_OK) {
        return ret;
    }

    if ((ret = hd_videoproc_open(HD_VIDEOPROC_0_IN_0, HD_VIDEOPROC_0_OUT_0,
&p_liveview_info->proc_path[0])) != HD_OK) {
        return ret;
    }
    if ((ret = hd_videoproc_open(HD_VIDEOPROC_1_IN_0, HD_VIDEOPROC_1_OUT_0,
&p_liveview_info->proc_path[1])) != HD_OK) {
        return ret;
    }
    if ((ret = hd_videoproc_open(HD_VIDEOPROC_2_IN_0, HD_VIDEOPROC_2_OUT_0,
&p_liveview_info->proc_path[2])) != HD_OK) {
        return ret;
    }

    if ((ret = hd_videoout_open(HD_VIDEOOUT_0_IN_0, HD_VIDEOOUT_0_OUT_0,
&p_liveview_info->out_path[0])) != HD_OK) {
        return ret;
    }
    if ((ret = hd_videoout_open(HD_VIDEOOUT_0_IN_1, HD_VIDEOOUT_0_OUT_0,
&p_liveview_info->out_path[1])) != HD_OK) {
        return ret;
    }
    if ((ret = hd_videoout_open(HD_VIDEOOUT_0_IN_2, HD_VIDEOOUT_0_OUT_0,
&p_liveview_info->out_path[2])) != HD_OK) {
        return ret;
    }

    if ((ret = hd_videoout_open(0, HD_VIDEOOUT_0_CTRL, &p_liveview_info->out_ctrl)) != HD_OK) { //open this
for device control
        return ret;
    }

    return ret;
}
```

```
HD_RESULT close_module(VIDEO_LIVEVIEW liveview_info)
{
   HD_RESULT ret = HD_OK;
   if ((ret = hd_videocap_close(liveview_info.cap_path[0])) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videocap_close(liveview_info.cap_path[1])) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videocap_close(liveview_info.cap_path[2])) != HD_OK) {
      return ret;
   }

   if ((ret = hd_videoproc_close(liveview_info.proc_path[0])) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_close(liveview_info.proc_path[1])) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_close(liveview_info.proc_path[2])) != HD_OK) {
      return ret;
   }

   if ((ret = hd_videoout_close(liveview_info.out_path[0])) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoout_close(liveview_info.out_path[1])) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoout_close(liveview_info.out_path[2])) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoout_close(liveview_info.out_ctrl)) != HD_OK) {
      return ret;
   }

   return ret;
}

HD_RESULT bind_module(void)
{
   HD_RESULT ret = HD_OK;

   //liveview - cap0
   if ((ret = hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOPROC_0_IN_0)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_bind(HD_VIDEOPROC_0_OUT_0, HD_VIDEOOUT_0_IN_0)) != HD_OK) {
      return ret;
   }

   //liveview - cap1
   if ((ret = hd_videocap_bind(HD_VIDEOCAP_0_OUT_1, HD_VIDEOPROC_1_IN_0)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_bind(HD_VIDEOPROC_1_OUT_0, HD_VIDEOOUT_0_IN_1)) != HD_OK) {
      return ret;
   }

   //liveview - cap2
   if ((ret = hd_videocap_bind(HD_VIDEOCAP_0_OUT_2, HD_VIDEOPROC_2_IN_0)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_bind(HD_VIDEOPROC_2_OUT_0, HD_VIDEOOUT_0_IN_2)) != HD_OK) {
      return ret;
   }

   return ret;
}
```

```
HD_RESULT unbind_module(void)
{
   HD_RESULT ret = HD_OK;

   //liveview - cap0
   if ((ret = hd_videocap_unbind(HD_VIDEOCAP_0_OUT_0)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_unbind(HD_VIDEOPROC_0_OUT_0)) != HD_OK) {
      return ret;
   }

   //liveview - cap1
   if ((ret = hd_videocap_unbind(HD_VIDEOCAP_0_OUT_1)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_unbind(HD_VIDEOPROC_1_OUT_0)) != HD_OK) {
      return ret;
   }

   //liveview - cap2
   if ((ret = hd_videocap_unbind(HD_VIDEOCAP_0_OUT_2)) != HD_OK) {
      return ret;
   }
   if ((ret = hd_videoproc_unbind(HD_VIDEOPROC_2_OUT_0)) != HD_OK) {
      return ret;
   }
   return ret;
}


HD_RESULT start_module(VIDEO_LIVEVIEW liveview_info)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videocap_start_list(liveview_info.cap_path, SAMPLE_LV_COUNT)) != HD_OK) {
      return ret;
   }

   if ((ret = hd_videoproc_start_list(liveview_info.proc_path, SAMPLE_LV_COUNT)) != HD_OK) {
      return ret;
   }

   if ((ret = hd_videoout_start_list(liveview_info.out_path, SAMPLE_LV_COUNT)) != HD_OK) {
      return ret;
   }

   return ret;
}


HD_RESULT stop_module(VIDEO_LIVEVIEW liveview_info)
{
   HD_RESULT ret = HD_OK;

   if ((ret = hd_videocap_stop_list(liveview_info.cap_path, SAMPLE_LV_COUNT)) != HD_OK) {
      return ret;
   }

   if ((ret = hd_videoproc_stop_list(liveview_info.proc_path, SAMPLE_LV_COUNT)) != HD_OK) {
      return ret;
   }

   if ((ret = hd_videoout_stop_list(liveview_info.out_path, SAMPLE_LV_COUNT)) != HD_OK) {
      return ret;
   }

   return ret;
}

int main(int argc, char** argv)
{
```

```
UINT32 key;
HD_RESULT ret;
VIDEO_LIVEVIEW liveview_info = {0};

//init memory
ret = hd_common_init(1);
if (ret != HD_OK) {
   printf("common init fail\n");
   return -1;
}
// init video cap/ video process/ video out module
ret = init_module();
if (ret != HD_OK) {
   printf("init fail\n");
   return -1;
}
// open video cap/ video process/ video out module
ret = open_module(&liveview_info);
if (ret != HD_OK) {
   printf("open fail\n");
   return -1;
}
// get lcd capability
ret = get_out_syscaps( liveview_info.out_ctrl, &liveview_info.lcd_syscaps);
if (ret != HD_OK) {
   printf("get video syscaps fail\n");
   return -1;
}
// set pip config
ret = set_pip_cfg( liveview_info);
if (ret != HD_OK) {
   printf("set video cfg fail\n");
   return -1;
}
// bind module
ret = bind_module();
if (ret != HD_OK) {
   printf("bind unit fail\n");
   return -1;
}
// start module
ret = start_module(liveview_info);
if (ret != HD_OK) {
   printf("start unit fail\n");
   return -1;
}

printf("\nLayout:\n");
printf("  +----------+----------+\n");
printf("  |          |          |\n");
printf("  |   LV0    |   LV1    |   LV0  : ch0\n");
printf("  | +-----+  |  +-----+ |   LV0' : ch0's second region\n");
printf("  | | LV0'|  |  | LV2 | |   LV1  : ch1 (on background layer)\n");
printf("  | +-----+  |  +-----+ |   LV2  : ch2 (on top layer)\n");
printf("  |          |          |\n");
printf("  |          |          |\n");
printf("  +----------+----------+\n");

while (1) {
   printf("Enter q to exit\n");
   key = getchar();
   if (key == 'q') {
      break;
   }
}
//stop module
ret = stop_module(liveview_info);
if (ret != HD_OK) {
   printf("stop unit fail\n");
   return -1;
}
```

```
    //unbind module
    ret = unbind_module();
    if (ret != HD_OK) {
        printf("unbind unit fail\n");
        return -1;
    }
    // close module
    ret = close_module(liveview_info);
    if (ret != HD_OK) {
        printf("close fail\n");
        return -1;
    }
    // uninit module
    ret = uninit_module();
    if (ret != HD_OK) {
        printf("uninit fail\n");
        return -1;
    }
    // uinit memory
    ret = hd_common_uninit();
    if (ret != HD_OK) {
        printf("common uninit fail\n");
        return -1;
    }

    return 0;
}
```

The control flow of Liveview with PIP sample code is described as below.

1. Initialize the modules used in Liveview with PIP, hd_videocap, hd_videoproc and hd_videoout.
2. Create the instance of the modules.
3. Get the video capability, such as the LCD resolution.
4. Set PIP config.
   a. Crop CH0 left-top 1/4 x 1/4 area, scale to 1/2 x 1 lcd size, then set to left side of background layer of videoout Path0.
   b. Scale down CH0 to 1/5 x 1/5 lcd size, then set to left side of top layer of videoout Path0.
   c. Scale down CH1 to 1/2 x 1 lcd size, then set to right side of background layer of videoout Path1.
   d. Crop CH2 left-top 1/4 x 1/4, scale to 1/5 x 1/5 of lcd size, then set to right side of top layer of videoout Path2
5. Change the status of the paths in the hd_videocap, hd_videoproc and hd_videoout instances to “Start”, and the stream flow will be start to run.
6. Check the specified key button from the console window, if “q” key is pressed, the stream flow will be closed.
7. Change the status of the paths in the hd_ videocap, hd_videoproc and hd_videoout instances to “Stop”.
8. Disconnect the instances used in the Liveview with PIP.
9. Close the instance of the modules.
10. Uninitialize the modules used in Liveview with PIP.

5.3. Multiple Paths Configuration

The following figure shows three stream paths in this multiple paths example code. The first path in yellow color outputs YUV frame to the *.yuv* file. The second path in red color connects hd_videocapture and hd_videoenc modules to output H.264 bit stream to *main.264* file. The last path in green color connects hd_videocapture, hd_videoprocess and hd_videoenc modules to output H.264 bit stream to *sub.264*. The main four stages of this example code: The first stage uses the APIs *hd_xxx_open()* to open the instance path. The second stage uses the APIs *hd_xxx_bind()* to connect each instance path together. The third stage uses *hd_videocap_pull_out_buf()* API to get YUV raw data, and the last stage uses *hd_videoenc_poll_list()* API to check the H.264 stream is available, if it is available, uses *hd_videoenc_recv_list()* API to get H.264 stream.

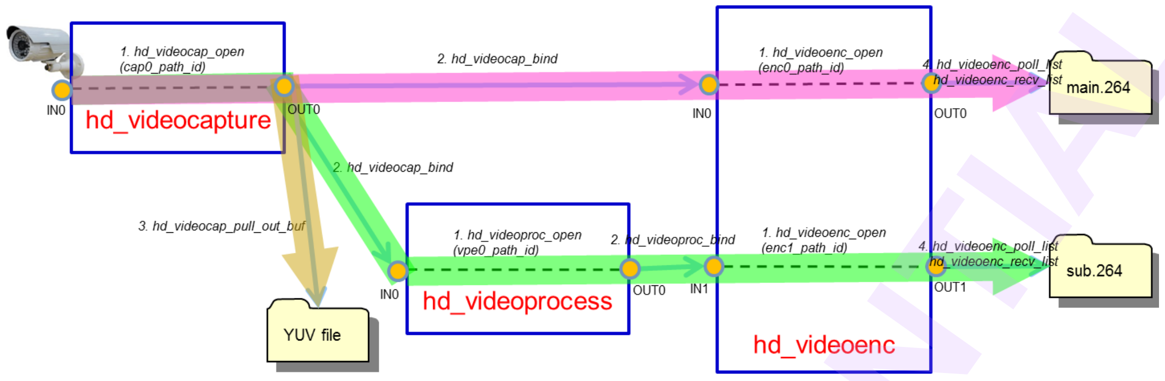

Fig. 11 HDAL Multiple Paths Diagram

Following is the source code of this multiple paths example.

This example code flow describes in the following:

```
/**
 * @file encode_with_substream.c
 * @brief encode two streams sample.
 * @author Schumy Chen
 * @date in the year 2018
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>
#include <sys/mman.h>
#include "hdal.h"

#define MAX_BITSTREAM_NUM 2
#define BITSTREAM_LEN      (720 * 576 * 3 / 2)

#define ALIGN64_DOWN(x)    (((x) >> 6) << 6)
#define ALIGN32_DOWN(x)    (((x) >> 5) << 5)
#define ALIGN16_DOWN(x)    (((x) >> 4) << 4)
#define ALIGN8_DOWN(x)     (((x) >> 3) << 3)
#define ALIGN4_DOWN(x)     (((x) >> 2) << 2)
#define ALIGN2_DOWN(x)     (((x) >> 1) << 1)

#define ALIGN64_UP(x)      ((((x) + 63) >> 6) << 6)
#define ALIGN32_UP(x)      ((((x) + 31) >> 5) << 5)
#define ALIGN16_UP(x)      ((((x) + 15) >> 4) << 4)
#define ALIGN8_UP(x)       ((((x) + 7) >> 3) << 3)
#define ALIGN4_UP(x)       ((((x) + 3) >> 2) << 2)
#define ALIGN2_UP(x)       ((((x) + 1) >> 1) << 1)

typedef struct _VIDEO_RECORD {
   HD_DIM  main_dim;
   HD_DIM  sub_dim;
   HD_PATH_ID cap0_path_id;
   HD_PATH_ID cap1_path_id;
   HD_PATH_ID vpe0_path_id;
   HD_PATH_ID enc0_path_id;
   HD_PATH_ID enc1_path_id;
   UINT32 enc_exit;
   INT test_item;
} VIDEO_RECORD;
```

```
HD_RESULT init_module(void)
{
   HD_RESULT ret;
   if((ret = hd_videocap_init()) != HD_OK)
      return ret;
   if((ret = hd_videoproc_init()) != HD_OK)
      return ret;
   if((ret = hd_videoenc_init()) != HD_OK)
      return ret;
   return HD_OK;
}

HD_RESULT open_module(VIDEO_RECORD *p_streams)
{
   HD_RESULT ret;
   if((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_0, &p_streams->cap0_path_id)) !=
HD_OK)
      return ret;
   if((ret = hd_videocap_open(HD_VIDEOCAP_0_IN_0, HD_VIDEOCAP_0_OUT_1, &p_streams->cap1_path_id)) !=
HD_OK)
      return ret;
   if((ret = hd_videoproc_open(HD_VIDEOPROC_0_IN_0, HD_VIDEOPROC_0_OUT_0, &p_streams->vpe0_path_id)) !=
HD_OK)
      return ret;
   if((ret = hd_videoenc_open(HD_VIDEOENC_0_IN_0, HD_VIDEOENC_0_OUT_0, &p_streams->enc0_path_id)) !=
HD_OK)
      return ret;
   if((ret = hd_videoenc_open(HD_VIDEOENC_0_IN_1, HD_VIDEOENC_0_OUT_1, &p_streams->enc1_path_id)) !=
HD_OK)
      return ret;
   return HD_OK;
}

HD_RESULT exit_module(void)
{
   HD_RESULT ret;
   if((ret = hd_videocap_uninit()) != HD_OK)
      return ret;
   if((ret = hd_videoproc_uninit()) != HD_OK)
      return ret;
   if((ret = hd_videoenc_uninit()) != HD_OK)
      return ret;
   return HD_OK;
}

HD_RESULT close_module(VIDEO_RECORD *p_streams)
{
   HD_RESULT ret;
   if((ret = hd_videocap_close(p_streams->cap0_path_id)) != HD_OK)
      return ret;
   if((ret = hd_videocap_close(p_streams->cap1_path_id)) != HD_OK)
      return ret;
   if((ret = hd_videoproc_close(p_streams->vpe0_path_id)) != HD_OK)
      return ret;
   if((ret = hd_videoenc_close(p_streams->enc0_path_id)) != HD_OK)
      return ret;
   if((ret = hd_videoenc_close(p_streams->enc1_path_id)) != HD_OK)
      return ret;
   return HD_OK;
}

static void *encode_thread(void *arg)
{
   VIDEO_RECORD* p_streams = (VIDEO_RECORD *)arg;
   INT i, pack_num, ret;
   INT ch = (INT)0;
   FILE *record_file[2];
   CHAR filename[50];
   CHAR *bitstream_data[MAX_BITSTREAM_NUM];
   HD_VIDEOENC_POLL_LIST poll_list[MAX_BITSTREAM_NUM];
   HD_VIDEOENC_RECV_LIST recv_list[MAX_BITSTREAM_NUM];
```

```
    // allocate buffer
    for (i = 0; i < MAX_BITSTREAM_NUM; i++) {
       bitstream_data[i] = (char *)malloc(BITSTREAM_LEN);
       if (bitstream_data[i] == 0) {
          return 0;
       }
       memset(bitstream_data[i], 0, BITSTREAM_LEN);
    }
     //----- open output files -----
    sprintf(filename, "ch%d_%lux%lu_main.265", ch, p_streams->main_dim.w, p_streams->main_dim.h);
    record_file[0] = fopen(filename, "wb");
    if (record_file[0] == NULL) {
       printf("open file error(%s)! \n", filename);
       exit(1);
    }

    sprintf(filename, "ch%d_%lux%lu_sub.265", ch, p_streams->sub_dim.w, p_streams->sub_dim.h);
    record_file[1] = fopen(filename, "wb");
    if (record_file[1] == NULL) {
       printf("open file error(%s)! \n", filename);
       exit(1);
    }

    memset(poll_list, 0, sizeof(poll_list));

    /*** Main stream */
    poll_list[0].path_id = p_streams->enc0_path_id;

    /*** sub substream */
    poll_list[1].path_id = p_streams->enc1_path_id;

    while (p_streams->enc_exit == 0) {
       /** poll bitstream until 500ms timeout */
       ret = hd_videoenc_poll_list(poll_list, MAX_BITSTREAM_NUM, 500);

       if (ret == HD_ERR_TIMEDOUT) {
          printf("Poll timeout!!");
          continue;
       }

       memset(recv_list, 0, sizeof(recv_list));
       for (i = 0; i < MAX_BITSTREAM_NUM; i++) {
          if (poll_list[i].revent.event != TRUE) {
             continue;
          }
          if (poll_list[i].revent.bs_size> BITSTREAM_LEN) {
             printf("bitstream buffer bs_size is not enough! %lu, %d\n",
                   poll_list[i].revent.bs_size, BITSTREAM_LEN);
             continue;
          }

          recv_list[i].path_id = poll_list[i].path_id;
          recv_list[i].user_bs.p_user_buf = bitstream_data[i];
          recv_list[i].user_bs.user_buf_size = BITSTREAM_LEN;
       }
         // receive stream data
       if ((ret = hd_videoenc_recv_list(recv_list, MAX_BITSTREAM_NUM)) < 0) {
          printf("Error return value %d\n", ret);
       } else {
          for (i = 0; i < MAX_BITSTREAM_NUM; i++) {
             if ((recv_list[i].retval < 0) && recv_list[i].path_id) {
                printf("path_id(%#lx): Error to receive bitstream. ret=%ld\n",
                      recv_list[i].path_id, recv_list[i].retval);
             } else if (recv_list[i].retval >= 0) {
                INT bs_size = 0;
                //write stream data
                for (pack_num = 0; pack_num < recv_list[i].user_bs.pack_num; pack_num++) {
                   bs_size += fwrite((CHAR *)recv_list[i].user_bs.video_pack[pack_num].user_buf_addr, 1,
                         recv_list[i].user_bs.video_pack[pack_num].size, record_file[i]);
                   fflush(record_file[i]);
                }
                printf("<CH%d, size=%d, frame_type=%d, newbsflag=0x%lx, timestamp=0x%lx>\n",
                      i, bs_size,
                      recv_list[i].user_bs.frame_type,
                      recv_list[i].user_bs.newbs_flag,
                      recv_list[i].user_bs.timestamp);
             }
          }
       }
    }
     // close output file
    for (i = 0; i < MAX_BITSTREAM_NUM; i++) {
       fclose(record_file[i]);
         //free buffer
       free(bitstream_data[i]);
    }
    return 0;
 }

 struct map_info {
    void *va;
    unsigned int map_size;
 };

 void *mmap_pa_to_va(unsigned int pa, unsigned int size, struct map_info *va_map_info, int mem_fd)
 {
    int page_size, map_size;
    unsigned int aligned_pa;
    void *mmap_va;

    if (pa == 0 || size == 0) {
       printf("Invalid param\n");
       return NULL;
    }
    page_size = getpagesize();
    aligned_pa = (pa / page_size) * page_size;
    map_size = (size + pa - aligned_pa);
    mmap_va = mmap(0, map_size, (PROT_READ | PROT_WRITE), MAP_SHARED, mem_fd, (int)aligned_pa);
    if (mmap_va == MAP_FAILED) {
       printf("mmap pa to va fail>\n");
       return NULL;
    }
    if (va_map_info) {
       va_map_info->map_size = map_size;
       va_map_info->va = mmap_va;
    }

    return mmap_va + pa - aligned_pa;
 }


 void save_output(char *filename, void *data, int size)
 {
    FILE *pfile;
    if ((pfile = fopen(filename, "wb")) == NULL) {
       printf("[ERROR] Open File %s failed!!\n", filename);
       exit(1);
    }
    fwrite(data, 1, size, pfile);
    fclose(pfile);
    printf("Write file: %s\n", filename);
 }

 HD_RESULT pull_out_buffer(HD_PATH_ID cap_path_id, int mem_fd)
 {
    HD_RESULT ret;
    HD_VIDEO_FRAME out_buffer;
    void  *out_buffer_va = NULL;
    struct map_info out_va_map_info;
    char filename[40];
    INT buf_size;
    static int idx = 0;

    /* Pull out buffer */
    ret = hd_videocap_pull_out_buf(cap_path_id, &out_buffer, 500);
    if (ret != HD_OK) {
       printf("hd_videocap_pull_out_buf fail\n");
       goto exit;
    } else {
       buf_size = hd_common_mem_calc_buf_size((void *)&out_buffer);
       out_buffer_va = mmap_pa_to_va((unsigned int)out_buffer.phy_addr[0], buf_size, &out_va_map_info,
 mem_fd);
       sprintf(filename, "encode_pull_out_%ldx%ld_idx%d.yuv", out_buffer.dim.w, out_buffer.dim.h, idx++);
       save_output(filename, out_buffer_va, buf_size);
    }

    /* Release out buffer */
    munmap(out_va_map_info.va, out_va_map_info.map_size);
    ret = hd_videocap_release_out_buf(cap_path_id, &out_buffer);
    if (ret != HD_OK) {
       printf("hd_videocap_release_out_buf fail\n");
       goto exit;
    }

 exit:
    return ret;
 }

 HD_RESULT set_path_config(HD_PATH_ID cap_path_id, HD_PATH_ID enc_path_id)
 {
    HD_VIDEOENC_PATH_CONFIG enc_config;
    HD_VIDEOCAP_PATH_CONFIG cap_config;
    HD_RESULT ret = HD_OK;

    //set videocap out pool
    memset(&cap_config, 0x0, sizeof(HD_VIDEOCAP_PATH_CONFIG));
    cap_config.data_pool[0].mode = HD_VIDEOCAP_POOL_ENABLE;
    cap_config.data_pool[0].ddr_id = 0;
    cap_config.data_pool[0].counts = HD_VIDEOCAP_SET_COUNT(4, 0);
    cap_config.data_pool[0].max_counts = HD_VIDEOCAP_SET_COUNT(4, 0);

    cap_config.data_pool[1].mode = HD_VIDEOCAP_POOL_DISABLE;
    cap_config.data_pool[2].mode = HD_VIDEOCAP_POOL_DISABLE;
    cap_config.data_pool[3].mode = HD_VIDEOCAP_POOL_DISABLE;

    ret = hd_videocap_set(cap_path_id, HD_VIDEOCAP_PARAM_PATH_CONFIG, &cap_config);
    if (ret != HD_OK) {
       printf("hd_videocap_set for main pool fail\n");
       return ret;
    }
    //set encode out pool
    memset(&enc_config, 0, sizeof(enc_config));
    enc_config.data_pool[0].mode = HD_VIDEOENC_POOL_ENABLE;
    enc_config.data_pool[0].ddr_id = 0;
    enc_config.data_pool[0].counts = HD_VIDEOENC_SET_COUNT(4, 0);
    enc_config.data_pool[0].max_counts = HD_VIDEOENC_SET_COUNT(4, 0);
    enc_config.data_pool[1].mode = HD_VIDEOENC_POOL_DISABLE;
    enc_config.data_pool[2].mode = HD_VIDEOENC_POOL_DISABLE;
    enc_config.data_pool[3].mode = HD_VIDEOENC_POOL_DISABLE;
    ret = hd_videoenc_set(enc_path_id, HD_VIDEOENC_PARAM_PATH_CONFIG, &enc_config);
    if (ret != HD_OK) {
       printf("hd_videoenc_set for main stream fail\n");
       return ret;
    }

    return ret;
 }

 HD_RESULT set_cap_param(HD_PATH_ID cap_path_id, HD_DIM dim, HD_VIDEO_PXLFMT pxlfmt)
 {
    HD_RESULT ret = HD_OK;
    HD_VIDEOCAP_OUT cap_out;
    //--- HD_VIDEOCAP_PARAM_OUT ---
    cap_out.dim.w = dim.w;
    cap_out.dim.h = dim.h;
    cap_out.pxlfmt = pxlfmt;
    ret = hd_videocap_set(cap_path_id, HD_VIDEOCAP_PARAM_OUT, &cap_out);
    if (ret != HD_OK) {
       printf("hd_videoenc_set for main stream fail\n");
       return ret;
    }

    return ret;
 }

 HD_RESULT set_enc_param(HD_PATH_ID enc_path_id, HD_DIM dim, HD_VIDEO_PXLFMT pxlfmt)
 {
    HD_VIDEOENC_OUT enc_param;
    HD_H26XENC_RATE_CONTROL rc_param;
    HD_VIDEOENC_IN video_in_param;
    HD_RESULT ret = HD_OK;
    //--- HD_VIDEOENC_PARAM_IN ---
    video_in_param.dim.w = dim.w;
    video_in_param.dim.h = dim.h;
    video_in_param.pxl_fmt = pxlfmt;
    ret = hd_videoenc_set(enc_path_id, HD_VIDEOENC_PARAM_IN, &video_in_param);
    if (ret != HD_OK) {
       printf("Error to set HD_VIDEOENC_PARAM_IN.\n");
       return ret;
    }

    //--- HD_VIDEOENC_PARAM_OUT_ENC_PARAM ---
    ret = hd_videoenc_get(enc_path_id, HD_VIDEOENC_PARAM_OUT_ENC_PARAM, &enc_param);
    enc_param.codec_type = HD_CODEC_TYPE_H265;
    enc_param.h26x.gop_num = 60;
    enc_param.h26x.svc_layer = HD_SVC_DISABLE;
    enc_param.h26x.profile = HD_H265E_MAIN_PROFILE;
    enc_param.h26x.level_idc = HD_H265E_LEVEL_4_1;
    enc_param.h26x.entropy_mode = HD_H265E_CABAC_CODING;
    ret = hd_videoenc_set(enc_path_id, HD_VIDEOENC_PARAM_OUT_ENC_PARAM, &enc_param);
    if (ret != HD_OK) {
       printf("Error to set HD_VIDEOENC_PARAM_OUT_ENC_PARAM.\n");
       return ret;
    }

    //--- HD_VIDEOENC_PARAM_OUT_RATE_CONTROL ---
    ret = hd_videoenc_get(enc_path_id, HD_VIDEOENC_PARAM_OUT_RATE_CONTROL, &rc_param);
    rc_param.rc_mode = HD_RC_MODE_CBR;
    rc_param.cbr.frame_rate_base = 30;   //fps = 30/1 = 30
    rc_param.cbr.frame_rate_incr = 1;
    rc_param.cbr.bitrate = 2 * 1024 * 1024;
    ret = hd_videoenc_set(enc_path_id, HD_VIDEOENC_PARAM_OUT_RATE_CONTROL, &rc_param);
    if (ret != HD_OK) {
       printf("Error to set HD_VIDEOENC_PARAM_OUT_RATE_CONTROL.\n");
       return ret;
    }

    return ret;
 }

 int main(int argc, char** argv)
 {
    HD_VIDEOCAP_SYSCAPS cap_syscaps;
    pthread_t enc_thread_id;

    HD_RESULT ret;
    VIDEO_RECORD streams = {0};
    INT mem_fd = -1;
    INT key;
    HD_VIDEO_PXLFMT pxlfmt = HD_VIDEO_PXLFMT_YUV420_NVX3;
    streams.enc_exit = 0;

    printf("Select encode substream by (0)Different Path  (1)Same Path\n");
    scanf("%d", &streams.test_item);

    /* Initialize mmap */
    if ((mem_fd = open("/dev/mem", O_RDWR | O_SYNC)) < 0) {
        printf("open /dev/mem failed.\n");
        return -1;
    }
     //open isf flow
    ret = hd_common_init(1);
    if(ret != HD_OK) {
        printf("init fail\n");
        goto exit;
    }
    //Initialize encode module
    ret = init_module();
    if(ret != HD_OK) {
        printf("init fail\n");
        goto exit;
    }
    //open video_record module
    ret = open_module(&streams);
    if(ret != HD_OK) {
        printf("open fail\n");
        goto exit;
    }

    //init dim for main/sub streams
    ret = hd_videocap_get(streams.cap0_path_id, HD_VIDEOCAP_PARAM_SYSCAPS, &cap_syscaps);
    if (ret != HD_OK) {
        printf("get syscaps fail\n");
        goto exit;
    }
    // set videoenc config (main)
    streams.main_dim.w  = cap_syscaps.max_dim.w;
    streams.main_dim.h  = cap_syscaps.max_dim.h;
    // set videoenc config (sub)
    streams.sub_dim.w  = cap_syscaps.max_dim.w / 2;
    streams.sub_dim.h  = cap_syscaps.max_dim.h / 2;

    //set main stream
    ret = set_path_config(streams.cap0_path_id, streams.enc0_path_id);
    if (ret != HD_OK) {
        printf("set_path_config 0 fail\n");
        goto exit;
    }
    // set videocap parameter (main)
    ret = set_cap_param(streams.cap0_path_id, streams.main_dim, pxlfmt);
    if (ret != HD_OK) {
        printf("set_cap_param fail\n");
        goto exit;
    }
    // set videoenc parameter (main)
    ret = set_enc_param(streams.enc0_path_id, streams.main_dim, pxlfmt);
    if (ret != HD_OK) {
        printf("set_enc_param fail\n");
        goto exit;
    }

    //set sub stream
    ret = set_path_config(streams.cap1_path_id, streams.enc1_path_id);
    if (ret != HD_OK) {
        printf("set_path_config 1 fail\n");
        goto exit;
    }
    // set videocap parameter (sub)
    ret = set_cap_param(streams.cap1_path_id, streams.sub_dim, pxlfmt);
    if (ret != HD_OK) {
        printf("set_cap_param fail\n");
        goto exit;
    }
    // set videoenc parameter (sub)
    ret = set_enc_param(streams.enc1_path_id, streams.sub_dim, pxlfmt);
    if (ret != HD_OK) {
        printf("set_enc_param fail\n");
        goto exit;
    }
```

```
//bind main stream: VIDEOIN(0) -> VIDEOENC(0)
ret = hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOENC_0_IN_0);
if (ret != HD_OK) {
   printf("videocap bind fail\n");
   goto exit;
}

if (streams.test_item == 0) {//Different Path
    //bind sub stream: VIDEOIN(1) -> VIDEOENC(1)
   hd_videocap_bind(HD_VIDEOCAP_0_OUT_1, HD_VIDEOENC_0_IN_1);
} else {                    //Same path
   //bind sub stream: VIDEOIN(0) -> VIDEOPROC(0) -> VIDEOENC(1)
   hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOPROC_0_IN_0);
   hd_videoproc_bind(HD_VIDEOPROC_0_OUT_0, HD_VIDEOENC_0_IN_1);
}

// bind video_record modules (main)
ret = hd_videocap_start(streams.cap0_path_id);
if (ret != HD_OK) {
   printf("start cap fail\n");
   goto exit;
}

ret = hd_videoenc_start(streams.enc0_path_id);
if (ret != HD_OK) {
   printf("start enc fail\n");
   goto exit;
}
// start video_record modules (sub)
if (streams.test_item == 0) {   //Different Path
   ret = hd_videocap_start(streams.cap1_path_id);
   if (ret != HD_OK) {
      printf("start cap1 fail\n");
      goto exit;
   }
} else {                            //Same path
   ret = hd_videoproc_start(streams.vpe0_path_id);
   if (ret != HD_OK) {
      printf("start vpe0 fail\n");
      goto exit;
   }
}
ret = hd_videoenc_start(streams.enc1_path_id);
if (ret != HD_OK) {
   printf("start enc fail\n");
   goto exit;
}
// create encode_thread (pull_out bitstream)
ret = pthread_create(&enc_thread_id, NULL, encode_thread, (void *)&streams);
if (ret < 0) {
   printf("create encode thread failed");
   goto exit;
}

// query user key
printf("Enter q to exit, p to pull capture out buffer\n");
while (1) {
   key = getchar();
   if (key == 'q') {
      // let encode_thread stop loop and exit
      streams.enc_exit = 1;
      // quit program
      break;
   } else if (key == 'p') {
      //pull outbuffer to yuv file
      ret = pull_out_buffer(streams.cap0_path_id, mem_fd);
      if (ret != HD_OK) {
         printf("pull out buffer fail\n");
         return -1;
      }
   }
```

```
    }
    // destroy encode thread
    pthread_join(enc_thread_id, NULL);
    // stop video_record modules (main)
    hd_videocap_stop(streams.cap0_path_id);
    hd_videoenc_stop(streams.enc0_path_id);
    // stop video_record modules (sub)
    if (streams.test_item == 0) {
       hd_videocap_stop(streams.cap1_path_id);
    } else {
       hd_videoproc_stop(streams.vpe0_path_id);
    }
    hd_videoenc_stop(streams.enc1_path_id);

    // unbind video_record module (main)
    hd_videocap_unbind(HD_VIDEOCAP_0_OUT_0);
    if (streams.test_item == 0) {
       // unbind video_record module (sub)
       hd_videocap_unbind(HD_VIDEOCAP_0_OUT_1);
    } else {
       // unbind video_record module (sub)
       hd_videoproc_unbind(HD_VIDEOPROC_0_OUT_0);
    }

exit:
    // close video_record module
    ret = close_module(&streams);
    if(ret != HD_OK) {
       printf("close fail\n");
    }
    // exit all modules
    ret = exit_module();
    if(ret != HD_OK) {
       printf("exit fail\n");
    }
    // uninit hdal
    ret = hd_common_uninit();
    if(ret != HD_OK) {
       printf("uninit fail\n");
    }
    // uninit memory
    if (mem_fd != -1) {
       close(mem_fd);
    }

    return 0;
}
```

1. Initialize the modules used in this multiple paths: hd_videocapture, hd_videoprocess and hd_videoenc.
2. Create the capture path, the video process path and the encode path.
3. Get the capture capability, such as the frame resolution.
4. Set the parameters of the encode path, such as the video input format and dimension, the codec type, GOP, the rate control mode, the freamrate and the bitrate.
5. Connect the paths, the main path is built by the following connection:
   hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOENC_0_IN_0);
   The outport #0 of the capture path connectes to the inport #0 of the encode path.
   The sub path is built by the following connections:
   hd_videocap_bind(HD_VIDEOCAP_0_OUT_0, HD_VIDEOPROC_0_IN_0);
   The outport #0 of the capture path connectes to the inport #0 of the video process path.
   hd_videoproc_bind(HD_VIDEOPROC_0_OUT_0, HD_VIDEOENC_0_IN_1);
   The outport #0 of the video process path connectes to the inport #1 of the encode path.
6. The stream flows start running after changing the status of the capture path, the video process path and the encode path to "Start".
7. Create a thread to use *hd_videoenc_poll_list()* API to poll the encoded stream is available or not, if yes, use *hd_videoenc_recv_list()* API to get the encoded stream, and then store them into the specified file.
8. Check the key button in the console window. In this sample code, we use 'p' key to get the YUV stream.

9. Press “q” key to close this sample code.
10. Change the status of the capture path, the video process path and the encode path to “Stop”.
11. Disconnect all connections.
12. Close all instance paths.
13. Uninitialize all modules.

6. Additional Functions

Please refer to HDAL design - NT9668x_NT98313 Design Specification documents.

7. HDAL API List

Please refer to HDAL design - NT9668x_NT98313 Design Specification documents.