Image Processing Unit V3 (IPUV3) Library

User's Guide

Document Number: 924-76397
Rev. 1.8
07/2010

ARM
POWERED®

freescale
semiconductor™

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Contents

Chapter 1 IPUV3 Library User's Guide

1.1   Introduction

This chapter presents IPU library related data types and APIs. IPU library is based on IPU hardware, it can implement below features:

- Resize
- Rotation
- Color space/format convert
- Overlay combination with the same size window which supports color key and alpha blending
- Output display to frame buffer directly after IPU process
- Windows crop
- Local alpha blending
- Video de-interlacing

IPU library assumes there are three kinds of operational buffers that could be in the IPU process:

- Input buffers, they contain the data which want to process, user can allocate by himself or let it be done by IPU library.
- Output buffers, they contain the data of finished process from input buffers, user can allocate by himself or let it be done by IPU library; if user wants to display output directly to frame buffer, then user does not need to allocate them, frame buffer now is the output buffer.
- Overlay buffers, they contain the data which want to process and combination.

Note:

The three buffers should be continuous.

There are two operation modes for IPU buffers:

- Stream mode, which will use double buffer in IPU low level operation.
- Normal mode, which will only use single buffer in IPU low level operation.

1.2 Example flow

This section lists some examples of IPU operations; all these examples can be tested by modifying `ipudev_config_file` file in IPU lib unit test:

- Input buffer: YUV format, QVGA
- Output buffer: RGB format, VGA
- IPU Lib: Resize, color space convert, rotation

Figure 1-1. Resize, CSC and rotation example

- Input buffer: RGB format, QVGA window in VGA buffers (enable input crop)
- Output buffer: RGB format, VGA
- IPU Lib: Resize, Input Crop
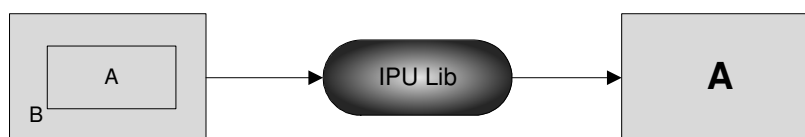
Figure 1-2. Input Crop example

- Input buffer: RGB format, VGA
- Output buffer: RGB format, QVGA window in VGA buffers (enable output crop)
- IPU Lib: Resize, Output Crop
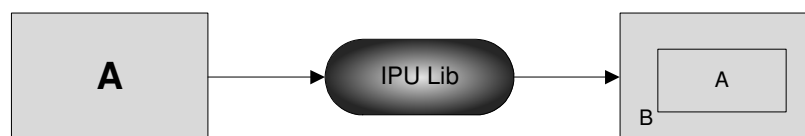
Figure 1-3. Output Crop example

- Input buffer: RGB565 format, QVGA
- Overlay buffer: BGR24 format, VGA
- Output buffer: RGB565 format, QVGA
- IPU Lib: Resize for overlay, format change for overlay, combination for input & overlay
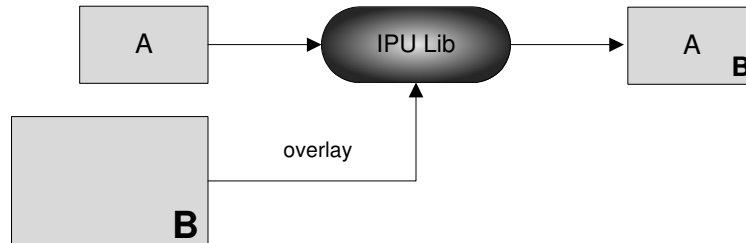
Figure 1-4. Overlay example

- Input buffers(double buffer): QVGA, index 1 & 2
- Output buffers (double buffer): VGA, index 1 & 2
- IPU Lib: stream mode, do resize for input.

Operation steps for double buffer, stream mode example:
- Prepare buffer A & B
- Finish index 1 – buffer A
- Finish index2 – buffer B
- Prepare buffer C
- Finish index1 – buffer C
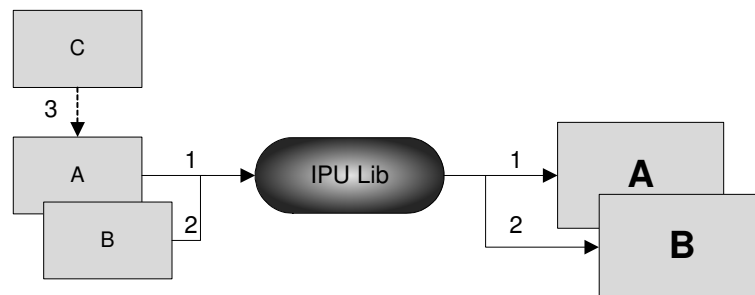
Figure 1-5. Double buffer, stream mode example

Operation steps for double buffer, stream mode, video de-interlace, low/medium motion example:

- Prepare buffer A , B & C
- Finish index 1 – buffer A
- Finish index 2 – buffer B
- Finish index 3 – buffer C
- Prepare buffer D
- Finish index1 – buffer D

Operation steps for single buffer, normal mode, video de-interlace, low/medium motion example:

- Prepare buffer A & B
- Finish index 1 – buffer A
- Finish index 2 – buffer B
- Prepare buffer C
- Finish index1 – buffer C

Operation steps for double buffer, stream mode, video de-interlace, high motion example:

- Prepare buffer A & B
- Finish index 1 – buffer A
- Finish index 2 – buffer B
- Prepare buffer C
- Finish index1 – buffer C

Operation steps for single buffer, normal mode, video de-interlace, high motion example:

- Prepare buffer A
- Finish index 1 – buffer A
- Prepare buffer B
- Finish index1 – buffer B

1.3   Source codes

The source codes of the IPU library are located into `imx-lib` LTIB package, refer to Table 1. To get its source code, run `./ltib -m prep -p imx-lib`

Then "`cd rpm/BUILD/imx-lib-<version>/ipu`"

Table 1. Source codes of the IPU library

File Name	Description
mxc_ipu_hl_lib.h	The headers file of IPU high-level library.
mxc_ipu_hl_lib.c	The source code of IPU high-level library
mxc_ipu_lib.c	The source code of IPU basic library implementation

1.4   Data types

1.4.1   Task mode

```
enum {
        TASK_ENC_MODE = 0x1,
        TASK_VF_MODE = 0x2,
        TASK_PP_MODE = 0x4,
        TASK_VDI_VF_MODE = 0x8,
        OP_NORMAL_MODE = 0x10,
        OP_STREAM_MODE = 0x20,
};
```

There are 3 time-sharing tasks in the IPU hardware: ENC, VF and PP. Those tasks are user selectable. TASK_VDI_VF_MODE, TASK_ENC_MODE and TASK_VF_MODE can not be used simultaneously.

User can choose OP_NORMAL_MODE for single buffer and OP_STREAM_MODE for double buffer mode.

1.4.2   Hardware task mode

```
enum {
        IC_ENC = 0x1,
        IC_VF = 0x2,
        IC_PP = 0x4,
        ROT_ENC = 0x8,
        ROT_VF = 0x10,
        ROT_PP = 0x20,
        VDI_IC_VF = 0x40,
};
```

These modes describe the exact IPU hardware task.

1.4.3   Control command

```
enum {
        IPU_CTL_ALLOC_MEM,
        IPU_CTL_FREE_MEM,
        IPU_CTL_TASK_QUERY,
        IPU_CTL_TASK_KILL,
};
```

User can call API mxc_ipu_lib_task_control() to do some special operation. See memory/task control parameter for details,

1.4.4   Memory control parameter

```
typedef struct {
        ipu_mem_info minfo;
        void * mmap_vaddr;
} ipu_lib_ctl_mem_t;
```

To allocate/free dma memory from ipu device :

```
ipu_lib_ctl_mem_t mem;
mem.minfo.size = size;
mxc_ipu_lib_task_control(IPU_CTL_ALLOC_MEM, (void *)(&mem), NULL);
 mxc_ipu_lib_task_control(IPU_CTL_FREE_MEM, (void *)(&mem), NULL);
```

1.4.5   Task control parameter

```
typedef struct {
        int index;
        int task_pid;
        int task_mode;
} ipu_lib_ctl_task_t;
```

To query/kill ipu task:

```
 ipu_lib_ctl_task_t task;
```

```
 task.index = query_index/kill_index;
mxc_ipu_lib_task_control(IPU_CTL_TASK_QUERY, (void *)(&task), NULL);
mxc_ipu_lib_task_control(IPU_CTL_TASK_KILL, (void *)(&task), NULL);
```

IPU task query could return special task's status including task pid and task mode, this task mode is described in 1.4.2.

IPU task kill is using to kill garbage tasks which may be killed by other process or segmentation fault, they have no chance to do de-init. User can check task_pid & task_mode to determine whether the task is a garbage one.

1.4.6   Input buffer parameter

```
typedef struct {
        unsigned int width;
        unsigned int height;
        unsigned int fmt;

     /* For VDI */
     ipu_motion_sel motion_sel;

     struct {
             struct mxcfb_pos pos;
             unsigned int win_w;
             unsigned int win_h;
     } input_crop_win;

        dma_addr_t user_def_paddr[3];
} ipu_lib_input_param_t;
```

These settings include input buffer's basic setting like width, height and fmt, it should be used FOURCC type to define format.

input_crop_win defines the input crop window from input buffer.

To allocate the input buffer, user_def_paddr parameter could be user allocated input buffer's physical address, if user do not define it (set it to NULL), ipu lib will allocate new buffers, and give out the virtual address through ipu_handle.inbuf_start, and for OP_STREAM_MODE, two user_def_paddr should be specified. Note that user_def_paddr[2] is used only for TASK_VDI_VF_MODE and motion mode is selected to be medium motion or low motion.

1.4.7   Overlay buffer parameter

```
typedef struct {
        unsigned int width;
        unsigned int height;
        unsigned int fmt;

        struct {
                struct mxcfb_pos pos;
                unsigned int win_w;
                unsigned int win_h;
        } ov_crop_win;

        dma_addr_t user_def_paddr[2];
        dma_addr_t user_def_alpha_paddr[2];

        unsigned char global_alpha_en;
        unsigned char local_alpha_en;
        unsigned char key_color_en;
        unsigned char alpha; /* 0 ~ 255*/
        unsigned int key_color; /* RBG 24bit */
} ipu_lib_overlay_param_t;
```

Similar process is followed for the input buffer parameter.

To get overlay it should be enabled at least one of the alpha blending or color key.

1.4.8   Output buffer parameter

```
typedef struct {
        unsigned int width;
        unsigned int height;
        unsigned int fmt;
```

```
        unsigned int rot;


        dma_addr_t user_def_paddr[3];


        int show_to_fb;
        struct {
                struct mxcfb_pos pos;
                unsigned int fb_num;
        } fb_disp;


        /* output_win is doing similar thing as fb_disp */
        /* they output data to part of the whole output */
        struct {
                struct mxcfb_pos pos;
                unsigned int win_w;
                unsigned int win_h;
        } output_win;
} ipu_lib_output_param_t;
```

The rot parameter defines the rotation that this output buffer should be done, the different rotation number represents:

```
        IPU_ROTATE_NONE = 0,
        IPU_ROTATE_VERT_FLIP = 1,
        IPU_ROTATE_HORIZ_FLIP = 2,
        IPU_ROTATE_180 = 3,
        IPU_ROTATE_90_RIGHT = 4,
        IPU_ROTATE_90_RIGHT_VFLIP = 5,
        IPU_ROTATE_90_RIGHT_HFLIP = 6,
        IPU_ROTATE_90_LEFT = 7,
```

Set show_to_fb to display output to frame buffer directly, if so, user can set fb_disp to choose the frame buffer device that wants to display (fb_num) and the display position in the primary display device.

If user does not enable show_to_fb, then user can define output_win to do output window crop.

In stream mode or after user enabled show_to_fb, the output buffer number is increased to 3 to avoid tearing issue. If user want to define the output buffer, please define all 3 buffers in these cases.

1.4.9   IPU task handle

```
typedef struct {
        void * inbuf_start[3];
        void * ovbuf_start[2];
        void * ovbuf_alpha_start[2];
        void * outbuf_start[3];
        int ifr_size;
        int ovfr_size;
        int ovfr_alpha_size;
        int ofr_size;

        void * priv;
} ipu_lib_handle_t;
```

This `handle` will be returned after `mxc_ipu_lib_task_init` function call. If user does not define `user_def_paddr` of input/overlay/output buffer, then user can get virtual address of input/overlay/output buffer by `inbuf_start/ovbuf_start/outbuf_start` which is allocated by IPU library.

The `ifr_size/ovfr_size/ofr_size` indicates the size of input/overlay/output buffer.

`priv` parameter should not be changed.

Note that inbuf_start[2] is used only for TASK_VDI_VF_MODE and motion mode is selected to be medium motion or low motion.

1.5   APIs

1.5.1   mxc_ipu_lib_task_init

```
/*!
 * This function init the ipu task according to param setting.
 *
 * @param      input        Input parameter for ipu task.
 *
 * @param      overlay      Overlay parameter for ipu task.
 *
 * @param      output       The output parameter for ipu task.
```

```
 *
 * @param      mode           The ipu mode user can define, refer to
 *                            header file.
 *
 * @param      ipu_handle     User just allocates this structure for init.
 *                            this parameter will provide some necessary
 *                            info after task init function.
 *
 * @return     This function returns 0 on success or negative error code on
 *             fail.
 */
int mxc_ipu_lib_task_init(ipu_lib_input_param_t * input,
                ipu_lib_overlay_param_t * overlay,
                ipu_lib_output_param_t * output,
                int mode, ipu_lib_handle_t * ipu_handle);
```

1.5.2   mxc_ipu_lib_task_update

```
/*!
* This function updates the buffer for special ipu task, it must be run after
* init function.
*
* For OP_STREAM_MODE mode, ipu task will take double buffer method, this
* function will return the next need-update buffer index number (0 or 1) on
* success, user should update input buffer according to it.
* Similar with it, output_callback's second parameter indicates the current
* output buffer index number(0 or 1), user should read output data from exact
* buffer according to it.
*
* For OP_NORMAL_MODE mode, ipu task will take single buffer method, so this
* function will always return 0 on success(next update buffer will keep on
* index 0), the same, output_callback's second parameter will keep on 0 too.
*
* How to update input buffer? If user has phys buffer themselves, please just
* update the phys buffer address by parameter phyaddr; if not, user can fill
* the input data to ipu_handle->inbuf_start[].
* For TASK_VDI_VF_MODE mode, if low motion or medium motion are used, user can
* not update the last used buffer's content, because the last used buffer is
* an aid buffer to generate the current de-interlaced frame.
*
```

```
* @param         ipu_handle      The ipu task handle need to update buffer.
*
* @param         new_inbuf_paddr User can set phyaddr to their own allocated
*                                 buffer addr, ipu lib will update the buffer
*                                 from this address for process. If user do not
*                                 want to use it, please let it be zero, and
*                                 fill the buffer according to inbuf_start
*                                 parameter in ipu_handle.
*
* @param         new_ovbuf_paddr User defined overlay physical buffer address.
*
* @param         new_ovbuf_alpha_paddr User defined overlay local alpha blending
*               physical buffer address.
*
* @param         output_callback IPU lib will call output_callback function
*                                 when there is output data.
*
* @param         output_cb_arg   The argument will be passed to output_callback.
*
* @return        This function returns the next update buffer index number on
*                success or negative error code on fail.
*/
int mxc_ipu_lib_task_buf_update(ipu_lib_handle_t * ipu_handle,
        dma_addr_t new_inbuf_paddr, dma_addr_t new_ovbuf_paddr,
        dma_addr_t new_ovbuf_alpha_paddr, void (output_callback)(void *, int),
        void * output_cb_arg);
```

1.5.3   mxc_ipu_lib_task_uninit

```
/*!
 * This function uninit the ipu task for special ipu handle.
 *
 * @param         ipu_handle      The ipu task handle need to un-init.
 *
 * @return        This function returns 0 on success or negative error code on
 *                fail.
 */
void mxc_ipu_lib_task_uninit(ipu_lib_handle_t * ipu_handle);
```

1.5.4   mxc_ipu_lib_task_control

```
/*!
 * This function control the ipu task according to param setting.
 *
 * @param        ctl_cmd          Control cmd.
 *
 * @param        arg              The control argument.
 *
 * @param        ipu_handle       User just allocate this structure for init.
 *                                this parameter will provide some necessary
 *                                info after task init function.
 *
 * @return       This function returns 0 on success or negative error code on
 *               fail.
 */
int mxc_ipu_lib_task_control(int ctl_cmd, void * arg, ipu_lib_handle_t *
ipu_handle);
```

1.6   Programming guide

1.6.1   How to use IPU library

1. `mxc_ipu_lib_task_init()`. Call `mxc_ipu_lib_task_init()` function with user defined setting.

User could set input/overlay/output setting like width/height/format/input crop/output to frame buffer etc.

User can allocate input, overlay and output buffer by them (must be physically continuous), if buffers are allocated the `user_def_paddr` parameter must be set in `ipu_lib_input_param_t/ipu_lib_overlay_param_t/ipu_lib_output_param_t`.

For `OP_STREAM_MODE` mode, user should set both of `user_def_paddr[2]`, for `OP_NORMAL_MODE` mode user only needs set `user_def_paddr[0]`. If video-deinterlacing function is needed and low/medium motion is used, for `OP_STREAM_MODE` mode, user should set all of `user_def_paddr[3]`, for `OP_NORMAL_MODE` mode user only needs set

user_def_paddr[2]. But, if video-deinterlacing function is needed and high motion is used, for OP_STREAM_MODE mode, user should set both of user_def_paddr[2], for OP_NORMAL_MODE mode user only needs set user_def_paddr[0].

mxc_ipu_lib_task_init() will return inbuf_start/ovbuf_start/outbuf_start in ipu_handle if user did not set user_def_paddr, these are virtual buffer start addresses allocated by IPU lib.

User should fill input/overlay data into user_def_paddr or inbuf_start/ovbuf_start before call function mxc_ipu_lib_task_buf_update().

NOTE:

Overlay is a special function of IPU, which can combine input and overlay to one output based on alpha and color-key setting. Overlay's width/height should be the same as output. If user does not want to use overlay function, then just let this parameter to NULL.

2. mxc_ipu_lib_task_buf_update(). User should call mxc_ipu_lib_task_buf_update() function after finishing fill input/overlay data into input/overlay user_def_paddr (user allocated buffer) or inbuf_start/ovbuf_start (IPU lib allocated buffer).

At first time calling this update function, for OP_STREAM_MODE mode, user should fill data to both input buffer inbuf_start[2], for OP_NORMAL_MODE mode user only needs to fill inbuf_start[0]. If video-deinterlacing function is needed and low/medium motion is used, for OP_STREAM_MODE mode, user should fill data to all input buffer inbuf_start[3], for OP_NORMAL_MODE mode user only needs to fill inbuf_start[2]. But, if video-deinterlacing function is needed and high motion is used, for OP_STREAM_MODE mode, user should set both input buffer inbuf_start[2], for OP_NORMAL_MODE mode user only needs to fill inbuf_start[0]; next time calling this update function, user only needs to fill buffer according to the index return by mxc_ipu_lib_task_buf_update() last time.

Above method is using buffers allocated by IPU lib but buffers can also be user allocated:

NOTE: during the first call of mxc_ipu_lib_task_buf_update()function, the new_inbuf_paddr/new_ovbuf_paddr/new_ovbuf_alpha_paddr will not take effect, first update function call only enable the ipu task, will not update new buffer, the new buffer update should begin from second update function call.

User defined buffer queue example (`OP_STREAM_MODE` mode):

a) user allocates 5 physically continuous memory buffers: `paddr[0~4]`;

b) set `input.user_def_paddr[2]` as `paddr[0]` and `paddr[1]`;

c) call `mxc_ipu_lib_task_init()`;

d) fill input data to `paddr[0]` and `paddr[1]`;

e) call `mxc_ipu_lib_task_buf_update()`;

f) fill input data to `paddr[2]`;

g) call `mxc_ipu_lib_task_buf_update(..&paddr[2]..)`;

In `mxc_ipu_lib_task_buf_update()` function, IPU lib will call `output_callback(void *arg, int output_buf_index)` (if user sets this call back function in parameter) while there is output data, user could handle output data by `paddr[output_buf_index]/outbuf_start[output_buf_index]`.

3. `mxc_ipu_lib_task_uninit()`

User should   call `uninit`
function to   disable IPU
task.

```
                    ┌──────────────────┐
                    │   Parameters     │
                    │    setting       │
                    └──────────────────┘
                             │
                             ▼
                    ┌──────────────────┐
                    │ mxc_ipu_lib_task_init │
                    │   (parameters)   │
                    └──────────────────┘
                             │
                             ▼
                        ◇ Input buffer
                          ready? ◇
                             │
                             ▼
          ┌──────────────────┐      ┌──────────────────┐
          │ mxc_ipu_lib_task_update │──▶│      Call        │
          │    (new_buffer)  │      │ output_callback() if │
          └──────────────────┘      │    it exists     │
                             │      └──────────────────┘
                             ▼
                        ◇ 緣 inished? ◇
                NO           │
                           YES
                             ▼
                    ┌──────────────────┐
                    │ mxc_ipu_lib_task_uninit() │
                    └──────────────────┘
```

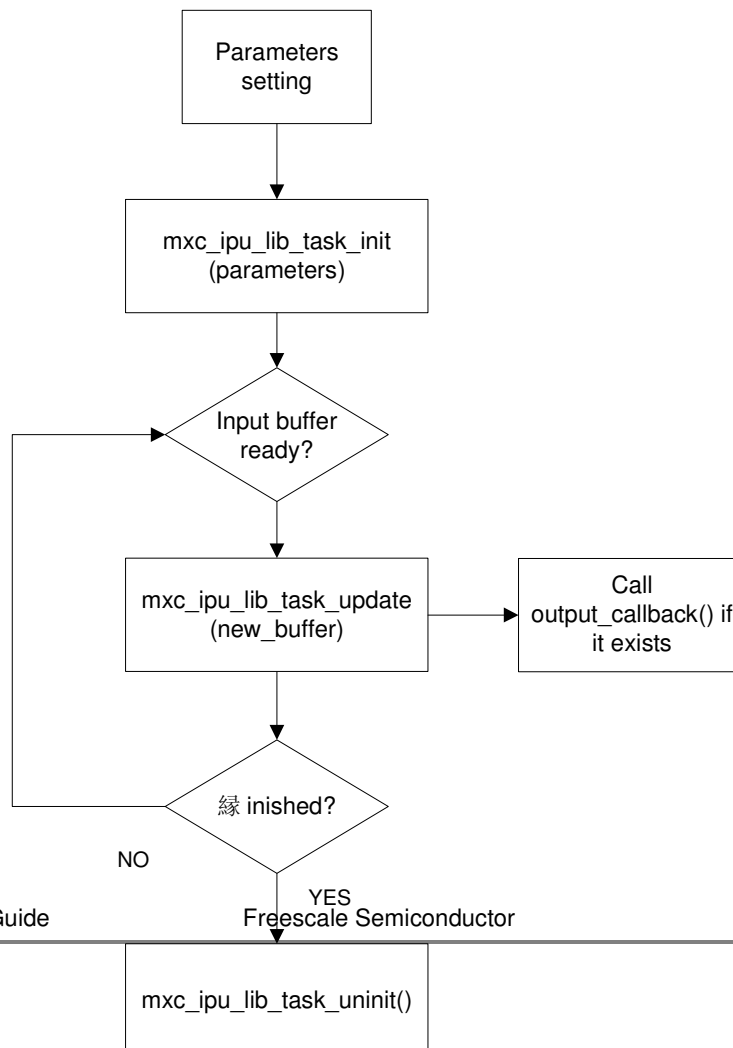**Figure 1-6. Simple calling flow of IPU Lib functions**

1.6.2   Unit test

Refer to `test/mxc_ipudev_test/mxc_ipudev_test.c`

    `test/mxc_ipudev_test/test_patterns.c`

    `test/mxc_ipudev_test/ipudev_config_file`

The usage of this unit test is shown below:

MXC IPU device Test

Usage: `./mxc_ipudev_test.out`

```
-C <config file>
-P <test pattern>
[-bw <block width for pattern 3>]
<input raw file>
```

test pattern:

1. video pattern with user defined dma buffer queue, one full-screen output

2. hopping block screen save

3. color bar + hopping block

4. color bar IC global alpha overlay

5. color bar IC separate local alpha overlay

6. color bar IC local alpha within pixel overlay

7. ipu dma copy test

8. 2 screen layer test using IC global alpha blending

9. 3 screen layer test using IC global alpha blending

10. 2 screen layer test using IC local alpha blending with alpha value in separate buffer

11. 3 screen layer test using IC local alpha blending with alpha value in separate buffer

12. 2 screen layer test using IC local alpha blending with alpha value in pixel

13. 3 screen layer test using IC local alpha blending with alpha value in pixel

14. 2 screen layer test IPC ProcessA + ProcessB with globla alpha blending

15. 2 screen layer test IPC ProcessA + ProcessB with local alpha blending

16. 3 screen layer test IPC ProcessA (first_layer + sencond_layer) + ProcessB (third_layer) with global alpha blending

17. 3 screen layer test IPC ProcessA (first_layer + sencond_layer) + ProcessB (third_layer) with local alpha blending

18. 3 screen layer test IPC ProcessA (first_layer) ProcessB (sencond_layer) ProcessC (third_layer) with local alpha blending

19. 2 screen layer test IPC ProcessA (first_layer) ProcessB (sencond_layer) with DP local alpha blending

As shown, there are 19 test patterns in `test_patterns.c`. And for other tests, user can also modify `ipudev_config_file`, for example:

The example cmd:

```
# ./mxc_ipudev_test –C ipudev_config_file qvga.yuv
```

The example config file:

```
#####   ipu dev test config file ##########
#
# fourcc ref:
#       RGB565->RGBP
#       BGR24 ->BGR3
#       RGB24 ->RGB3
#       BGR32 ->BGR4
#       BGRA32->BGRA
#       RGB32 ->RGB4
#       RGBA32->RGBA
#       ABGR32->ABGR
#       YUYV  ->YUYV
#       UYVY  ->UYVY
#       YUV444->Y444
#       NV12  ->NV12
#       YUV420P->I420
#       YUV422P->422P
#       YVU422P->YV16
#
# rotation ref:
#       IPU_ROTATE_NONE = 0,
#       IPU_ROTATE_VERT_FLIP = 1,
#       IPU_ROTATE_HORIZ_FLIP = 2,
#       IPU_ROTATE_180 = 3,
#       IPU_ROTATE_90_RIGHT = 4,
#       IPU_ROTATE_90_RIGHT_VFLIP = 5,
#       IPU_ROTATE_90_RIGHT_HFLIP = 6,
#       IPU_ROTATE_90_LEFT = 7,
#
# mode ref:
#       TASK_ENC = 0x1
#       TASK_VF = 0x2
#       TASK_PP = 0x4
#       TASK_VDI_VF = 0x8
#       NORMAL_MODE = 0x10
#       STREAM_MODE = 0x20
#
# video de-interlace motion ref:
#       MEDIUM_MOTION = 0
#       LOW_MOTION = 1
```

```
#       HIGH_MOTION = 2
#

#### mode
mode=0x24

#### operation frame count
fcount=50

#### operation loop count
loop_cnt=1

#### input
in_width=320
in_height=240
in_fmt=I420
#input crop
in_posx=0
in_posy=0
in_win_w=0
in_win_h=0
#input motion(video de-interlace)
motion_sel=0

#### output
out_width=1024
out_height=768
out_fmt=UYVY
out_rot=0
#output to framebuffer
out_to_fb=1
out_fb_num=2
out_posx=0
out_posy=0
#output to file
out_filename=output.dat
```

This example uses VF task, stream mode, input file is `qvga.yuv`, and input parameters are 320x240@I420, output parameters are 1024x768@UYVY, the output should feed to `framebuffer2`.

Chapter 2 Screen layer library user guide

2.1 Introduction

This chapter presents the screen layer library which is based on IPU library previously described.

The screen layer library:
- Provides user space API to support multi-layer GUI/Video display on Linux platform.
- Provides user space API to support hardware acceleration for image process including color space conversion, resize, rotation, alpha blending, color key etc. by IPU.
- Provides an abstract layer to add more hardware acceleration for device support.

2.2 Data flow
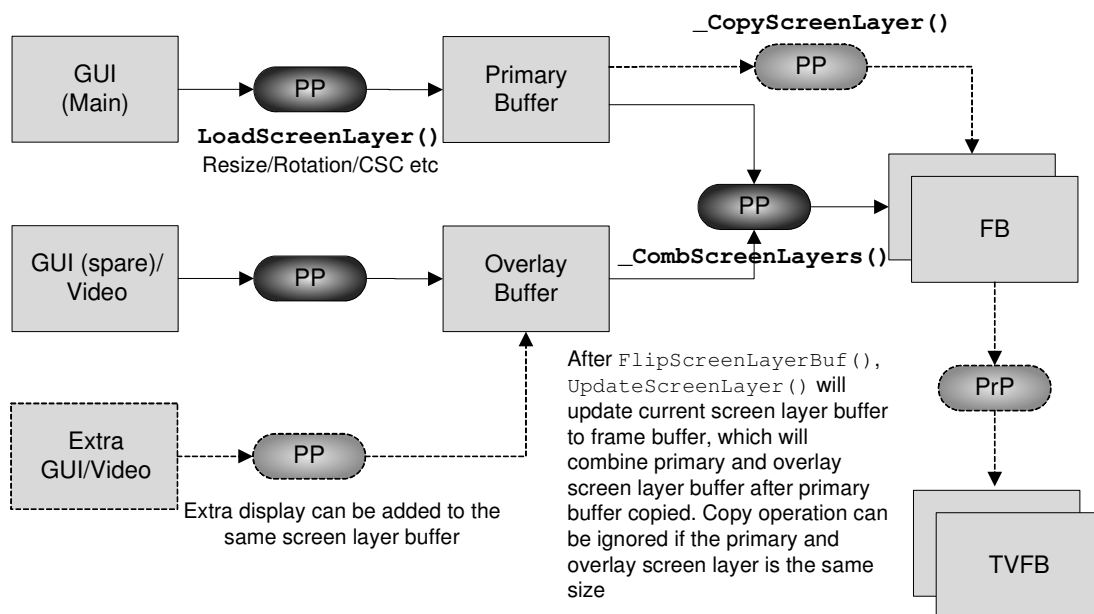
Figure 2-1. Screen Layer lib data flow

2.3   Source codes

The source codes of screen layer library are located into `imx-lib` LTIB package, refer to Table 2. To get the source code, run:

```
./ltib –m prep –p imx-lib
```

Then "`cd rpm/BUILD/imx-lib-<version>/ screenlayer`"

Table 2. Source codes of the screen layer library

File Name	Description
ScreenLayer.h	The header files of screen layer library.
ScreenLayer.c	The source code of screen layer library

2.4   Data types

2.4.1   Ret code

```
typedef enum {
        E_RET_SUCCESS = 0,
        E_RET_DEV_FAIL,
        E_RET_WRONG_FMT,
        E_RET_MEM_ALOC_FAIL,
        E_RET_MMAP_FAIL,
        E_RET_PRIMARY_ERR,
        E_RET_RECT_OVERFLOW,
        E_RET_BUFIDX_ERR,
        E_RET_TASK_SETUP_ERR,
        E_RET_TASK_RUN_ERR,
        E_RET_FLIP_ERR,
        E_RET_NOSUCH_METHODTYPE,
        E_RET_DESTORY_PRI_WITH_SUBSL,
        E_RET_ALPHA_BLENDING_CONFLICT,
        E_RET_LOCAL_ALPHA_BLENDING_DISABLE,
        E_RET_ALPHA_BUF_NOT_ALLOC_ERR,
        E_RET_IPC_SEM_OPEN_FAILED,
        E_RET_IPC_SHM_FAILED,
} SLRetCode;
```

This `enum` define the return code of screen layer APIs.

2.4.2   Screen rectangle

```
typedef struct {
      u16    left;
      u16    top;
      u32    right;
      u32    bottom;
} SLRect;
```

2.4.3   Screen layer

```
typedef struct {
        SLRect          screenRect;
        u32             fmt;
        u32             bufSize;
        u32             bufAlphaSize;
        bool            supportSepLocalAlpha;
        void         ** bufVaddr;
        dma_addr_t    * bufPaddr;
        void         ** bufAlphaVaddr;
        dma_addr_t    * bufAlphaPaddr;
        void          * pPrimary;
        char            fbdev[32];
        u32             flag;
        void          * pPriv;
} ScreenLayer;
```

This is the main structure of the screen layer library, user should set some parameters before creating a screen layer.

To create primary screen layer, user must set pPrimary to NULL, and do not need to set screenRect , it is decided by fbdev user set.

To create overlay screen layer, user must set its primary screen layer to pPrimary, the screenRect defines the screen rectangle in its primary screen layer.

If user set bufPaddr to NULL, screen layer library will allocate screen layer buffers in create function, bufVaddr is the virtual address of the allocated buffer list, bufPaddr is the physical address of the allocated buffer list.

If user wants to use buffers allocated by other application, just set `bufPaddr` to his allocated buffer list.

If user enables separate local alpha, `bufAlphaVaddr` is valid for the virtual address of the allocated alpha buffer list, `bufAlphaPaddr` is the physical address of the allocated alpha buffer list.

The same as `bufPaddr`, user can set `bufAlphaPaddr` if he wants to use alpha buffers allocated by other application.

2.4.4   Load parameter

```
typedef struct {
        u32             srcWidth;
        u32             srcHeight;
        u32             srcFmt;
        SLRect          srcRect;
        SLRect          destRect;
        u32             destRot;
        dma_addr_t      srcPaddr;
} LoadParam;
```

This structure is used in `LoadScreenLayer` function; it defines source window and destination window settings.

User must set source buffer's physical address to `srcPaddr` for `LoadScreenLayer`.

User can set `srcRect` to do input crop in a source window (`srcWidth`, `srcHeight`).

User can also set `destRect` to output a spare window in `dest` screen layer window.

2.4.5   Method setting parameter

```
typedef enum {
        E_SET_ALPHA,
        E_SET_COLORKEY,
        E_ENABLE_LAYER,
        E_COPY_TVOUT,
} SetMethodType

typedef struct {
```

```
       u8      globalAlphaEnable;
       u8      sepLocalAlphaEnable;
       u32     alpha;
} MethodAlphaData;


typedef struct {
       u8      enable;
       u32     keyColor;
} MethodColorKeyData;


typedef struct {
       u8      tvMode;
       u32     lcd2tvRotation;
} MethodTvoutData;
```

These parameters are used to set overlay, user can set local/global alpha and color key.

User can enable copy to TV by `MethodTvoutData`, this parameter should be set only for the screen layer which is being updated.

2.5　APIs

2.5.1　CreateScreenLayer

```
SLRetCode CreateScreenLayer(ScreenLayer *pSL, u8 nBufNum);
```

This function creates one screen layer based on `pSL`'s setting described above.

For one screen layer, it can contain more than one buffer, the define `nBufNum` to create more buffers for this screen layer.

2.5.2　LoadScreenLayer

```
SLRetCode LoadScreenLayer(ScreenLayer *pSL, LoadParam *pParam, u8 nBufIdx);
```

This function fills a screen layer with source image; it will store the result in the buffers allocated by `CreateScreenLayer()` which can also be accessible by user, `nBufIdx` specifies the store buffer index.

IPU will do resize and color space conversion and rotation according to the `pParam`.

2.5.3   UpdateScreenLayer

```
SLRetCode UpdateScreenLayer(ScreenLayer *pSL);
```

This function updates the current buffer of the screen layer to display the frame buffer, the frame buffer is decided by `fbdev` setting during `CreateScreenLayer()`.

When the function updates primary layer, it will also update overlay buffer to LCD. When function updates overlay layer, primary layer will be automatically updated at the same time.

2.5.4   FlipScreenLayerBuf

```
SLRetCode FlipScreenLayerBuf(ScreenLayer *pSL, u8 nBufIdx);
```

This function will set the current buffer of the specified screen layer.

2.5.5   SetScreenLayer

```
SLRetCode SetScreenLayer(ScreenLayer *pSL, SetMethodType eType, void
*setData);
```

This function sets screen layer's alpha and color key.

2.5.6   DestoryScreenLayer

```
SLRetCode DestoryScreenLayer(ScreenLayer *pSL);
```

2.6 Unit test

Refer to `test/mxc_ipudev_test/mxc_ipudev_test.c`.

`test/mxc_ipudev_test/test_patterns.c (screenlayer_test).`

Chapter 3 IPU DP module combination

3.1 Introduction

DP is a hardware module in IPU which can do CSC and combination for two frame buffers.

The combination can be done through global alpha or local alpha, both implement by the frame buffer driver.

3.2 Combination IOCTL for fb driver

3.2.1 Definition to alpha structures

```
struct mxcfb_gbl_alpha {
        int enable;
        int alpha;
};


struct mxcfb_loc_alpha {
        int enable;
        unsigned long alpha_phy_addr0;
        unsigned long alpha_phy_addr1;
};
```

3.2.2 DP global alpha combination

To enable DP global alpha combination feature, we need to use `fb ioctl`
`MXCFB_SET_LOC_ALPHA`. A variable in `struct mxcfb_gbl_alpha` type then is needed. The
argument `d` of `ioctl` must be an open file descriptor of `/dev/fb*`. By default, `/dev/fb0` stands
for the background frame buffer and `/dev/fb2` stands for the foreground frame buffer, that
means only `/dev/fb0` and `/dev/fb2` are valid for this `ioctl`. If the open file descriptor of
`/dev/fb0` is passed to this `ioctl`, the graphics plane of DP is set to the background plane,
otherwise, the graphics plane is set to the foreground plane. The graphics plane always gets the

alpha value no matter the alpha type is global or local. If the alpha value is bigger, the graphics plane shows more clearly. In the case of the open file descriptor of `/dev/fb0` is passed to this `ioctl`, and the global alpha value is set to be 255, then the background plane shows itself over the foreground plane.

For example, the following code will enable DP global alpha feature and shows both background plane and the foreground plane:

```
struct mxcfb_gbl_alpha g_alpha;
                g_alpha.alpha = 128;
                g_alpha.enable = 1;
                if (ioctl(fd_fb, MXCFB_SET_GBL_ALPHA, &g_alpha) < 0) {
                        printf("Set global alpha failed\n");
                        close(fd_fb_0);
                        return TFAIL;
                }
```

The argument `fd_fb` is an open file descriptor of `/dev/fb*`.

3.2.3   DP local alpha combination (alpha value is contained in separate buffer)

Similar to DP global alpha combination feature, it is needed to enable DP local alpha combination by calling some `fb ioctl` APIs. The main `ioctl` APIs are `MXCFB_SET_LOC_ALPHA` and `MXCFB_SET_LOC_ALP_BUF`. The `ioctl MXCFB_SET_LOC_ALPHA` is similar to `MXCFB_SET_LOC_ALPHA`. Only `/dev/fb0` and `/dev/fb2` are valid for this `ioctl`.

The graphics plane of DP is determined by the open file descriptor argument. Although the graphics plane can be set to the background plane or the foreground plane, only foreground plane is tested to be set to the graphics plane. The `ioctl` of `MXCFB_SET_LOC_ALPHA` enables the DP local alpha combination and tells the user the physical addresses of the two alpha buffers.

The user needs to do memory map for the two buffers and gets the virtual addresses of the two buffers so that the user can fill the buffers with specific alpha value. The `ioctl` of `MXCFB_SET_LOC_ ALP_BUF` choose one of the two alpha buffers to be active by passing the physical address of the alpha buffer which is returned from the `ioctl` of the

MXCFB_SET_LOC_ALPHA. It is recommended to change the alpha buffer number every time when the user needs to update the alpha buffer.

To enable DP local alpha combination feature with the graphics plane set to be the foreground plane, it is needed to follow these steps:

1. Call the fb ioctl of FBIOPUT_VSCREENINFO to set the var info of the foreground frame buffer:

```
fb2_var.xres = g_display_width;
fb2_var.yres = g_display_height;
fb2_var.xres_virtual = g_display_width;
fb2_var.yres_virtual = g_display_height*2;
if (ioctl(fd_fb_2, FBIOPUT_VSCREENINFO,
         &fb2_var) < 0) {
         printf("Put var of fb2 failed\n");
         close(fd_fb_2);
         return TFAIL;
         }
```

2. Call the fb ioctl of MXCFB_SET_LOC_ALPHA to enable DP local alpha combination feature and get the physical address of the two local alpha buffers:

```
l_alpha.enable = 1;
l_alpha.alpha_phy_addr0 = 0;
l_alpha.alpha_phy_addr1 = 0;
if (ioctl(fd_fb_2, MXCFB_SET_LOC_ALPHA,
         &l_alpha) < 0) {
         printf("Set local alpha failed\n");
         close(fd_fb_2);
         return TFAIL;
}
loc_alpha_phy_addr0 =
         (unsigned long)(l_alpha.alpha_phy_addr0);
loc_alpha_phy_addr1 =
         (unsigned long)(l_alpha.alpha_phy_addr1);
```

3. Memory map the two local alpha buffers so that the user can set the specific alpha value:

```
        alpha_buf_size = fb2_var.xres * fb2_var.yres;


        alpha_buf0 = (char *)mmap(0, alpha_buf_size,
                   PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd_fb_2,
                   loc_alpha_phy_addr0);
        if ((int)alpha_buf0 == -1) {
                   printf("\nError: failed to map alpha buffer 0"
                         " to memory.\n");
                   close(fd_fb_2);
                   return TFAIL;
        }
        alpha_buf1 = (char *)mmap(0, alpha_buf_size,
                   PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd_fb_2,
                   loc_alpha_phy_addr1);
        if ((int)alpha_buf1 == -1) {
                   printf("\nError: failed to map alpha buffer 1"
                         " to memory.\n");
                   munmap((void *)alpha_buf0, alpha_buf_size);
                   close(fd_fb_2);
                   return TFAIL;
        }
```

4. Fill the alpha buffer with specific alpha value and call the `fb ioctl` of `MXCFB_SET_LOC_ALP_BUF` to choose one of the two alpha buffers to be the active alpha buffer. It is recommended to use ping-pang buffer mode:

```
/* The window shows graphics and video planes. */
fill_alpha_buffer(alpha_buf0, 0, 0,
g_display_width, g_display_height, 0x80);


if (ioctl(fd_fb_2, MXCFB_SET_LOC_ALP_BUF, &loc_alpha_phy_addr0) < 0) {
    printf("Set local alpha buf failed\n");
    close(fd_fb_2);
    return TFAIL;
}
```

```
sleep(5);

fill_alpha_buffer(alpha_buf1, 0, 0,
     g_display_width, g_display_height, 0x80);

if (ioctl(fd_fb_2, MXCFB_SET_LOC_ALP_BUF, &loc_alpha_phy_addr1) < 0) {
    printf("Set local alpha buf failed\n");
    close(fd_fb_2);
    return TFAIL;
}
```

3.3   Unit test

The user can test the DP local alpha combination feature (with alpha value contained in separate buffer) by the following tests:

1. V4L2 overlay unit test:

   ```
   /unit_tests/mxc_v4l2_overlay.out –ow 240 –oh 320 –ol 20 –ot 20 –a 1 –d 2
   –fg
   ```

2. IPU library unit test:

   ```
   /unit_tests/mxc_ipudev_test.out –P 20
   ```