Application Report
SPRACG2–May 2018

Sharing VPE Between VISIONSDK and PSDKLA

Marvin Liang and Somnath Mukherjee

ABSTRACT

The video processing engine (VPE) is a hardware module in the Jacinto/TDA family of devices that mainly provides deinterlace, scaler and color space conversion. Most of the ADAS customers use VPE to process SD camera deinterlacing for RVC or SRV use cases. Typically, the development is based on TI VISION SDK. Most infotainment customers use VPE to process the scaling or color space conversion for played multimedia video. Typically, the development is based on TI PSDKLA.

Project collateral and source code discussed in this document can be downloaded from the following URL: http://www.ti.com/lit/zip/spracg2.

Contents

Trademarks

All trademarks are the property of their respective owners.

1 VPE Share Problem in Current Use Cases

More customers are considering a system that supports infotainment features plus SRV in one single Jacinto/TDA. Since only one Jacinto/TDA has one instance of VPE, there is a driver conflict for these use cases. This application report provides an implementation to solve the driver conflict problem when simultaneously sharing the VPE module with different SDKs.

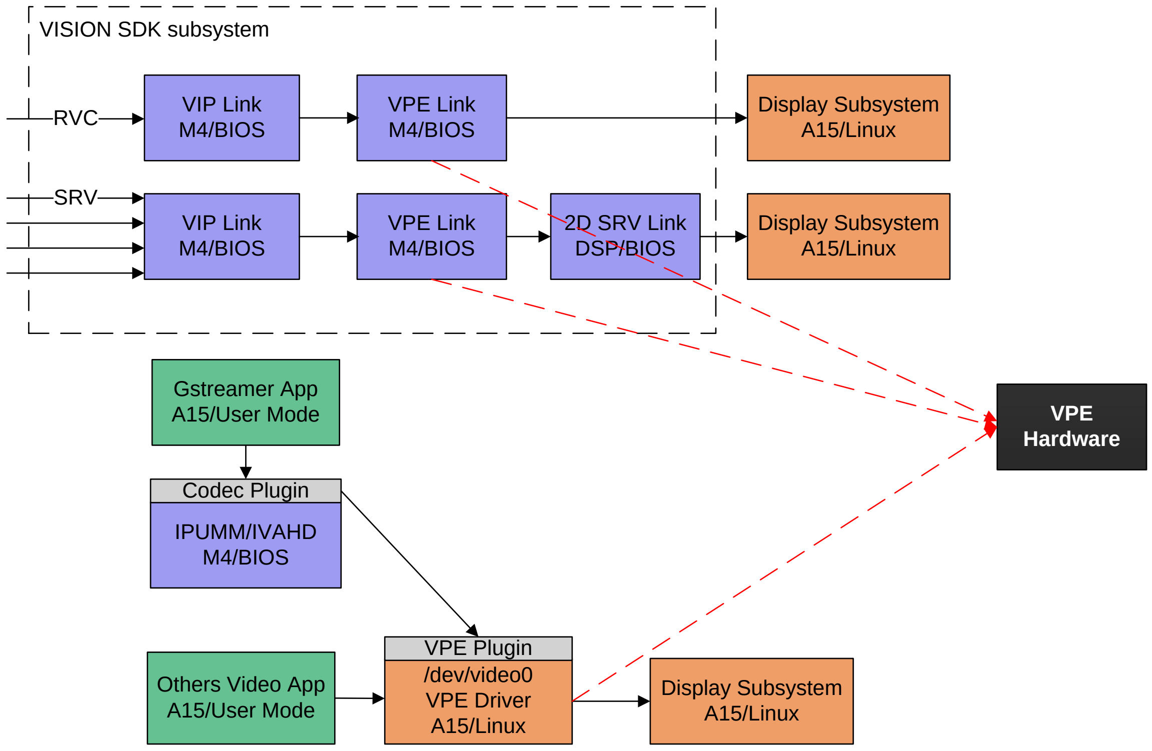


Figure 1. VPE Share Problem Between VISION SDK and PSDKLA

Consider a system that supports these features as shown in Figure 1:

- RVC or SRV with SD cameras

  For fast boot consideration, most customers choose VISION SDK framework for developing RVC or SRV. VISION SDK designs with the Link and Chain concept are used to implement any use case. When the SD camera frames (typically, YUV420) captured by VIP link, VIP sends the frames to the VPE link for deinterlacing. The VPE link calls the driver based on the TI PDK, which is located in the M4 core to configure the VPE hardware module to implement the deinterlacing, and then sends the interlaced frames to next link.
- Multimedia player

  By default, TI PSDKLA uses open source gstreamer framework for any multimedia player use case. TI provides gstreamer plugins, for example ducatiH264, ducaitiMJPEG, ducatiVPE. These plugins can work in gstreamer pipeline to process the multimedia file. For example, play a 720x480 H.264 video and display it on a 1280x720 LCD. The gstreamer calls the ducatiH264 plugin to configure IPUMM on the M4 core to communicate with IVAHD to decode the H.264 frames to YUV420. Then, it sends the information to the VPE plugin, to upscaling, to 1280x720, which can be displayed on 1280x720 LCD with full screen mode. The VPE plugin calls the TI Linux VPE V4L2 driver, which is located on the A15 core, to configure the VPE hardware module to implement the upscaling. Then, it sends the up scaled frames to the next plugin.

If A and B use cases occur simultaneously, both cases will fail because one single VPE hardware instance cannot be controlled or configured from both M4 and A15 cores. Actually, any other VPE share use case that needs both the PDK driver on M4 and Linux driver on A15 will fail. For example, the VISION SDK subsystem needs a CSC except when the SD camera requires the deinterlace; PSDKLA needs a CSC also.

A new mechanism is needed to share the VPE hardware module between VISION SDK and PSDKLA.

2 VPE Sharing Design

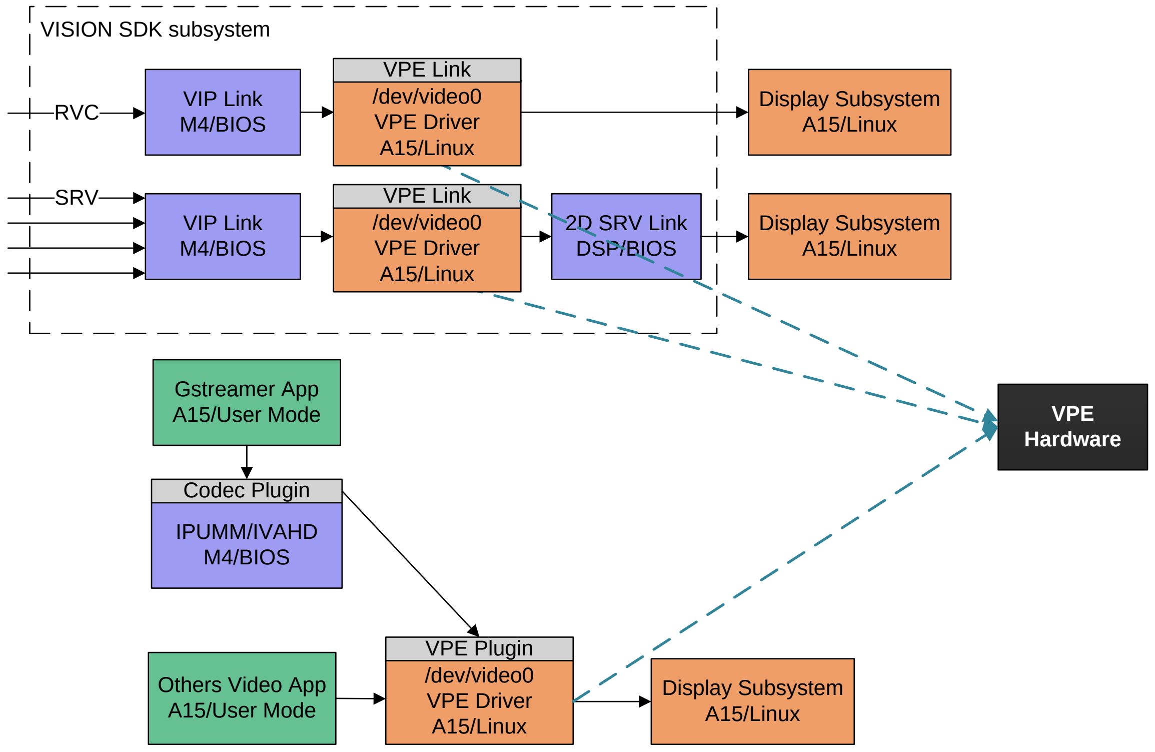

VISION SDK subsystem
RVC
SRV
VIP Link M4/BIOS
VIP Link M4/BIOS
VPE Link /dev/video0 VPE Driver A15/Linux
VPE Link /dev/video0 VPE Driver A15/Linux
2D SRV Link DSP/BIOS
Display Subsystem A15/Linux
Display Subsystem A15/Linux
VPE Hardware
Gstreamer App A15/User Mode
Codec Plugin
IPUMM/IVAHD M4/BIOS
Others Video App A15/User Mode
VPE Plugin /dev/video0 VPE Driver A15/Linux
Display Subsystem A15/Linux

Figure 2. New VPE Sharing Concept

Sharing the VPE hardware requires that the driver can only be called in ONE core: M4 or A15. If the gstreamer VPE plugin calls the M4-based VPE driver, the plugin needs more software effort to implement the IPC communication and buffer sharing between SYSBIOS and Linux. The better way is redesigned: a new VPE link in VISION SDK that includes these characteristics as shown in Figure 2.

- The new VPE link follows the VISION SDK Link and Chain framework.
- The new VPE link calls the A15-based Linux V4L2 driver to implement all VPE related features.
- All of the exchanged buffers between links are allocated from VISION SDK heap. These buffers should be shared with SYSBIOS and Linux.

3 VPE Sharing Implementation

3.1 Construct DMABUF in Linux

The VISION SDK subsystem uses the TI SYSBIOS OS. The drivers that include VPE on M4 can access the physical address directly. All of the buffers shared between VISION SDK links on the M4 core are described by physical address. But, Linux has a different mechanism to access a buffer by driver. DMABUF is a basic data structure or descriptor for a buffer in Linux that can share information between the Linux drivers. The buffer that is allocated from the VISION SDK heap and shared with the Linux VPE driver needs to be converted to a DMABUF file descriptor. These steps describe the construction of a DMABUF fd that can be used for the Linux V4L2 driver according to the address of the VPE buffer in VISION SDK heap:

1. Open the driver /dev/vmemexp that is the standard feature in TI K4.4.
2. Construct a DMABUF fd by system call DBUFIOC_EXPORT_VIRTMEM with the buffer's virtual address and size.

3. Reference sample code:

```
Int32 A15VpeLink_drvOpen(A15VpeLink_ChObj *pChObj)
{
    Int8 devname[20] = "/dev/video0";
    vpe_params *vpe = &pChObj->vpePrm;

    vpe->fd =  open(devname, O_RDWR);
        if(vpe->fd < 0)
                pexit("Cant open %s\n", devname);

        printf("vpe:%s open success!!!fd %d\n", devname,vpe->fd);
     devBufFD = open("/dev/vmemexp", O_RDWR | O_CLOEXEC);
    return vpe->fd;
}
static Int32 A15VpeLink_drvExportDmaBuf(void * vAddr, uint32_t size, uint32_t *fdBuf)
{
    int retVal = -1;
    struct dmabuf_vmem_export exp;
    exp.vaddr = (unsigned long)vAddr;
    exp.size = size;

    if(devBufFD > 0)
    {
        /* Export as DMAbuf handle */
        retVal = ioctl(devBufFD, DBUFIOC_EXPORT_VIRTMEM, &exp);
        if(retVal == 0)
        {
            *fdBuf = exp.fd;
        }
        else
        {
            printf(" exportDmaBuf failed \n ");
        }
    }

    return retVal;
}
Int32 A15VpeLink_drvConstructDmabuf(Int32 drm_fd,Int32 *handle_buf_dmafd,Int32
*handle_uv_buf_dmafd,Int32 *buf_dmafd,Int32 *uv_buf_dmafd,System_Buffer *pBuffer)
{
    System_VideoFrameBuffer *pVideoBuf=NULL;
    struct drm_prime_handle req;
    UInt32 size_Y=0,size_UV=0;
    Int8 ret;

    pVideoBuf = (System_VideoFrameBuffer *)pBuffer->payload;
    A15VpeLink_drvVideoFrameGetSize(&pVideoBuf->chInfo,&size_Y,&size_UV);

    ret = A15VpeLink_drvExportDmaBuf(((UInt32)pVideoBuf->bufAddr[0], size_Y,&buf_dmafd);
    if (ret) {
        return ret;//failed
    }


    if(uv_buf_dmafd && size_UV)
{
        ret = A15VpeLink_drvExportDmaBuf(((UInt32)pVideoBuf-
>bufAddr[1], size_UV,& uv_buf_dmafd);
        if (ret) {
            return ret;//failed
        }
    }

    return SYSTEM_LINK_STATUS_SOK;
```

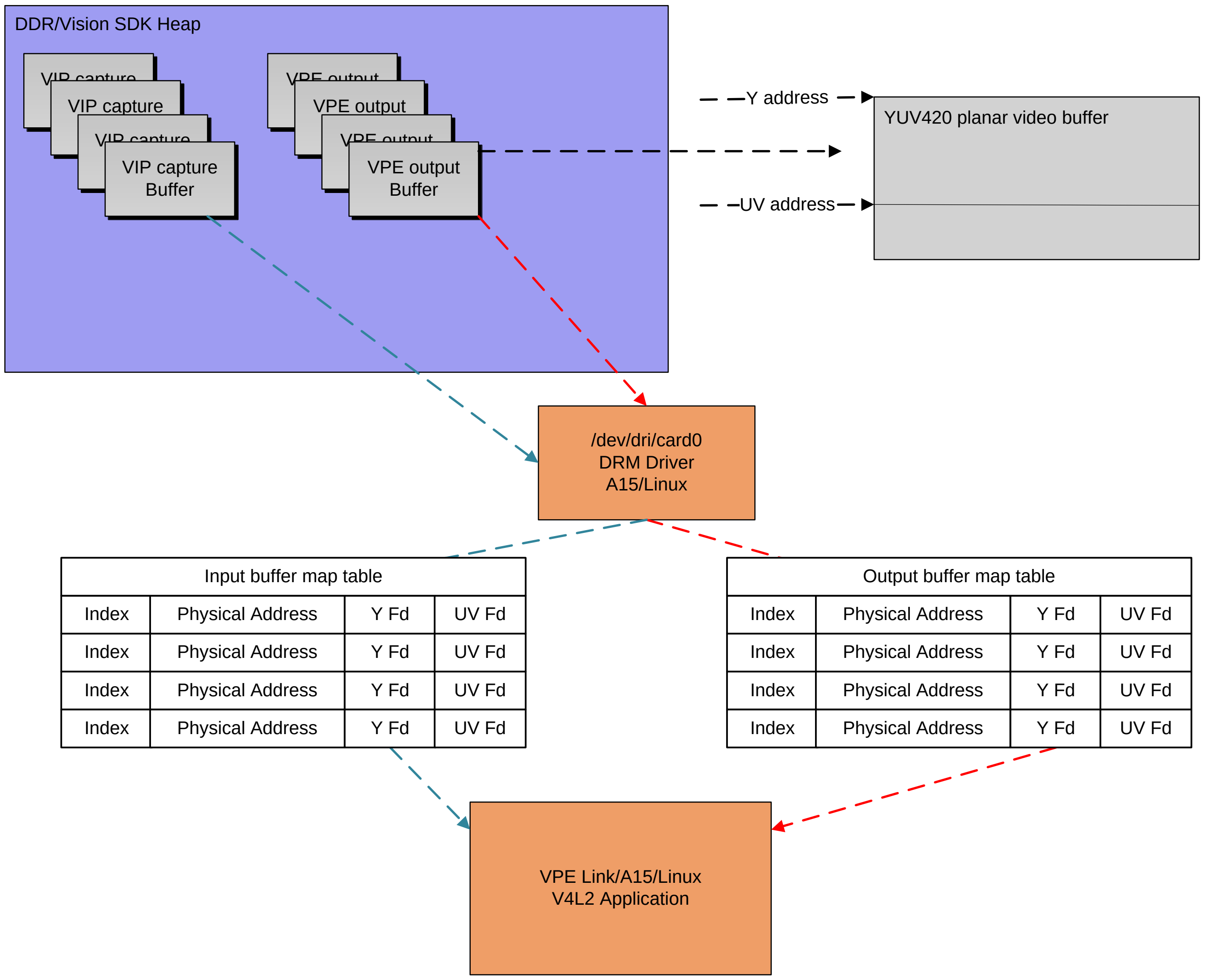
3.2 VPE Buffer Sharing Between TI SYSBIOS and Linux

When the VPE V4L2 link is initialized, it allocates the output buffers from the VISION SDK heap. After each buffer is allocated, it is constructed to a DMABUF with a corresponding file descriptor fd. An output buffer mapping table has been created to record the relationship between this buffer's physical address and the DMABUF fd (see Figure 3).

When VPE gets the camera data or input buffer from the VIP link, it is constructed to a DMABUF with a corresponding file descriptor fd. An Input buffer mapping table has been created to record the relationship between this buffer's physical address and the DMABUF fd (see Figure 3).

The YUV420 planar format buffer needs to be constructed to two DMABUF fd: one for Y vector and one for UV vector. This is required by the VPE Linux driver.

This methodology is shown in Figure 3.


DDR/Vision SDK Heap
VIP capture
VIP capture Buffer
VPE output
VPE output Buffer
Y address
UV address
YUV420 planar video buffer
/dev/dri/card0 DRM Driver A15/Linux
VPE Link/A15/Linux V4L2 Application

Input buffer map table			
Index	Physical Address	Y Fd	UV Fd
Index	Physical Address	Y Fd	UV Fd
Index	Physical Address	Y Fd	UV Fd
Index	Physical Address	Y Fd	UV Fd

Output buffer map table			
Index	Physical Address	Y Fd	UV Fd
Index	Physical Address	Y Fd	UV Fd
Index	Physical Address	Y Fd	UV Fd
Index	Physical Address	Y Fd	UV Fd

Figure 3. VPE Buffer Sharing Between SYSBIOS and Linux

3.3 VPE V4L2 Link Main Processing

Figure 4 shows the main processing of the VPE V4L2 link. The important logic is when the data buffer is received from the previous link, the buffer address of the VISION SDK heap has to be converted to DMABUF. Only DMABUF can be used in V4L2 APIs of the VPE Linux driver. After the VPE V4L2 link gets the DMABUF of the new generated data and before the link sends it to the next link, the DMABUF has to be converted to the buffer address of VISION SDK heap.

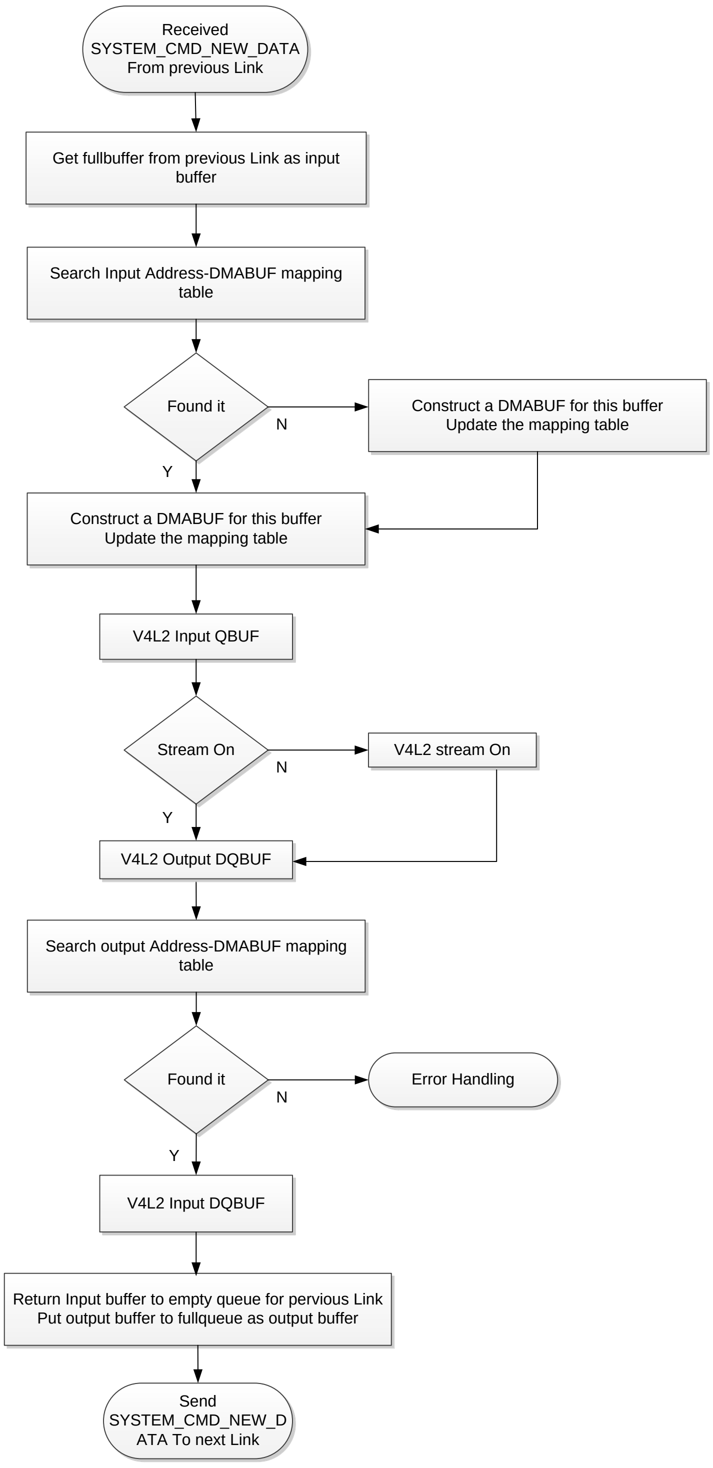

Received SYSTEM_CMD_NEW_DATA From previous Link
Get fullbuffer from previous Link as input buffer
Search Input Address-DMABUF mapping table
Found it
N
Construct a DMABUF for this buffer Update the mapping table
Y
Construct a DMABUF for this buffer Update the mapping table
V4L2 Input QBUF
Stream On
N
V4L2 stream On
Y
V4L2 Output DQBUF
Search output Address-DMABUF mapping table
Found it
N
Error Handling
Y
V4L2 Input DQBUF
Return Input buffer to empty queue for pervious Link Put output buffer to fullqueue as output buffer
Send SYSTEM_CMD_NEW_DATA To next Link

Figure 4. VPE V4L2 Link Main Process Flowchart

3.4 VISION SDK 2D SRV chain with VPE V4L2 link

After the VPE V4L2 link is implemented, the 2D SRV chain in VISION SDK can be updated as shown in Figure 5. The main difference between the original chain with the M4 driver VPE link and the new chain is that the VPE V4L2 link located in A15. This chain requires the associated IPCIN and IPCOUT links to convert the IPC message between M4 and A15.

NOTE: The IPCIN link in A15 automatically converts the physical address of the shared buffers to the virtual address. There is a need for the physical address when constructing the DMABUF file descriptor. You need to manually convert the buffer address (from the IPCIN link) from the virtual address to the physical address by calling the VISION SDK API.

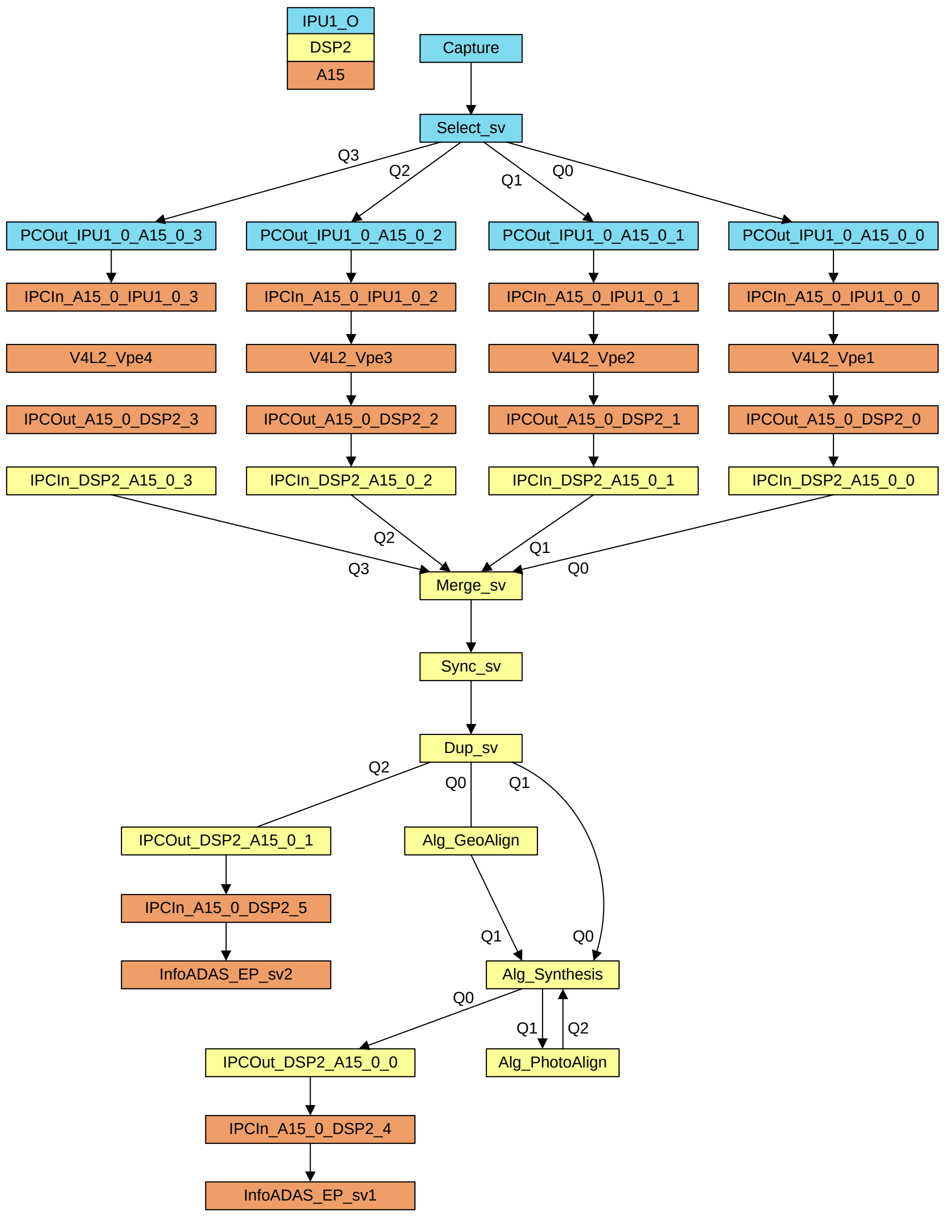


Figure 5. 2D SRV Chain With VPE V4L2 Link

4 References

- *Video Processing Engine* chapter of the *DRA75x, DRA74x SoC for Automotive Infotainment Silicon Revision 2.0, 1.1 Technical Reference Manual*
- *Exporting virtual memory as dmabuf*