Texas Instruments REAL WORLD SIGNAL PROCESSING™

BIOS PSP

VPIF Device Driver Design Document

Information in this document is subject to change without notice. Texas Instruments may have pending patent applications, trademarks, copyrights, or other intellectual property rights covering matter in this document. The furnishing of this document is given for usage with Texas Instruments products only and does not give you any license to the intellectual property that might be contained within this document. Texas Instruments makes no implied or expressed warranties in this document and is not responsible for the products based from this document

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

This document describes the VPIF DSP/BIOS™ device driver. The VPIF device driver design presented here is in the context of C6748/OMAPL138 SOCs targeted for various Video Applications using DSP/BIOS™ operating system running on C674x VLIW DSP. The VPIF driver conforms to the IOM driver model specified by the DSP/BIOS™ operating system.

⚐ *The usage of structure names and field names used throughout this design document is only for indicative purpose. These names shall not necessarily be matched with the names used in source code.*

□ *All references to OMAPL138 are also applicable to C6748. They are used interchangeably throughout this document.*

1.1 Purpose & Scope

The purpose of this document is to explain the device driver design for VPIF peripheral. This document provides details regarding the design of the VPIF driver and also the data types, data structures and application programming interfaces provided by the VPIF driver, the requirements it places on the software environment where it can be deployed, how to customize/ configure it to specific requirements etc. The target audience includes device driver developers from TI as well as consumers of the driver.

Although this document makes explicit reference to a typical DSP/BIOS™ GIO class device driver as an example, the general aspects of a device driver design presented here, are equally applicable to other class of devices and OS.

This document does not explain how to use the VPIF device driver, for usage instructions please refer to the BIOSPSP user guide.

1.2 Terms & Abbreviations

Term	Description
⚐	This bullet indicates important information. Please read such text carefully.
□	This bullet indicates additional information.
API	Application Programmer’s Interface
CC	Closed Caption
CGMS	Copy generation management system
CSL	TI Chip Support Library – primitive h/w abstraction
EDC	External Device Control
HD	High Definition
IOM	Input / Output Module
IP	Intellectual Property

ISR	Interrupt Service Routine
OMAPL138/C6748	TI's digital multi-media processor with C674x core
OS	Operating System
SD	Standard definition
SOC	System on chip
VPIF	Video Port Interface
WSS	Wide screen signaling

1.3 References

1.	spru403q.pdf	DSP/BIOS™ API Reference Guide
2.	xxxx_BIOSPSP_Userguide.pdf	VPIF user guide
3.	sprugj9.pdf	VPIF H/W Controller
4.	BIOSPSP_vpif.chm	VPIF chm
5.	BIOSPSP_vpifedc.chm	VPIF Edc chm

1.4 Overview

Video Port Interface provides a flexible video input/output port which allows the capture and display of digital video streams. The following sub section explains in detail the hardware and the software context of the VPIF driver.

1.4.1 Hardware Overview

VPIF module is used to receive and transmit video data on C6748/OMAPL138. The VPIF module has two input channels to receive video byte stream data and two output channels with which video byte stream can be asserted. Both input and output channels have the same architecture.

The VPIF Capture interface consists of one of the following interfaces:

- Maximum 2 channels of SD input in BT.656 format
- Maximum 1 channel of HD input in BT.1120 format
- Maximum 1 channel of RAW input, with data bit widths from 8 bits/pixel, 10 bits/pixel, and 12 bits/pixel mode.

The VPIF Display interface consists of one of the following interfaces:

- 2 Channel of SD output in BT.656 format
- 1 Channel of HD output in Rec.1120 format

The following figure shows the basic block diagram of VPIF.

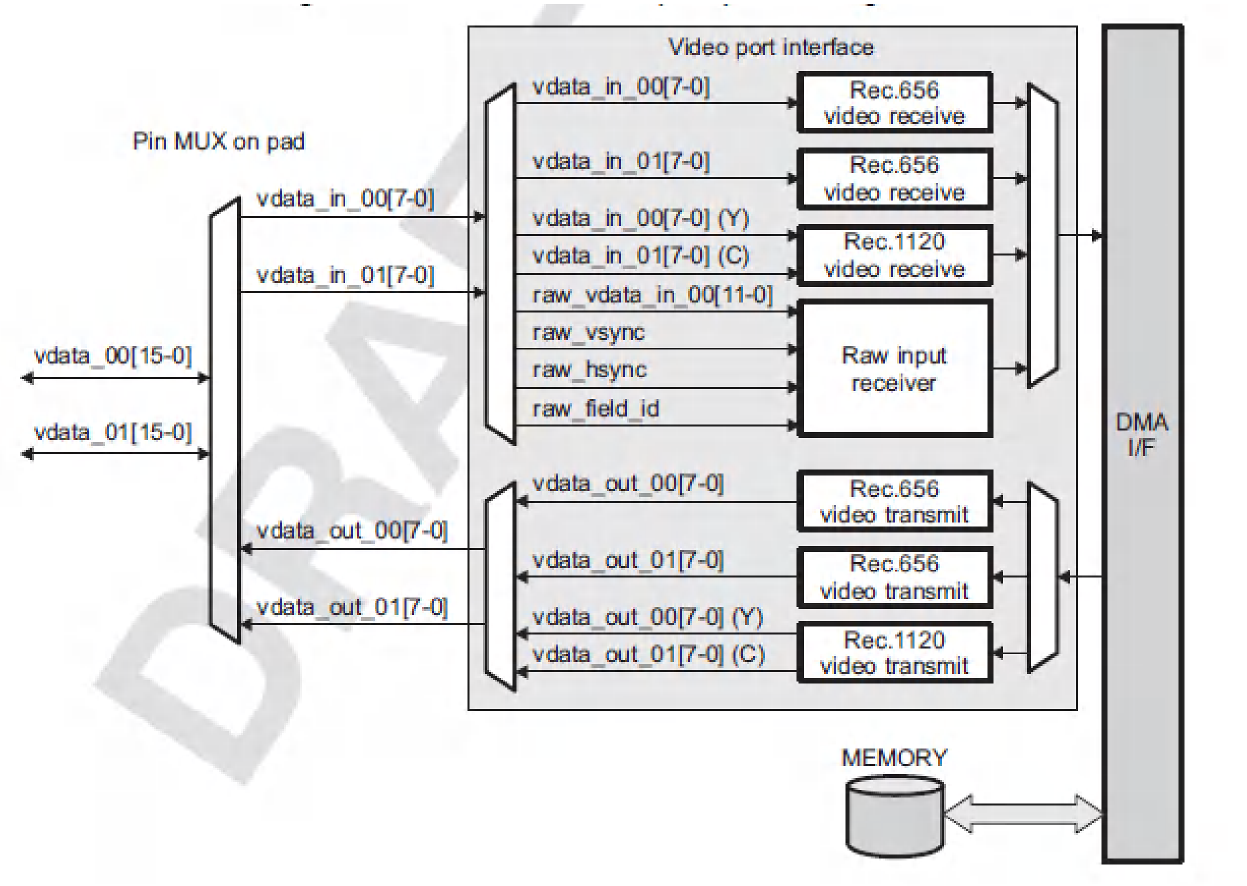


Figure 1. VPIF Hardware Overview

The following decoders are used for various types of captures:

- Two TVP5147 decoders are connected to both channels via BT.656 interface. One TVP5147 decoder is connected to S-video input which provides BT.656 input to channel1. The other TVP5147 decoder is connected to composite input which provides BT.656 input to channel0.
- External MT9T001 sensor is connected to both the channels for RAW data capture.

The following figure shows the physical connections for TVP5147 decoders on EVM.

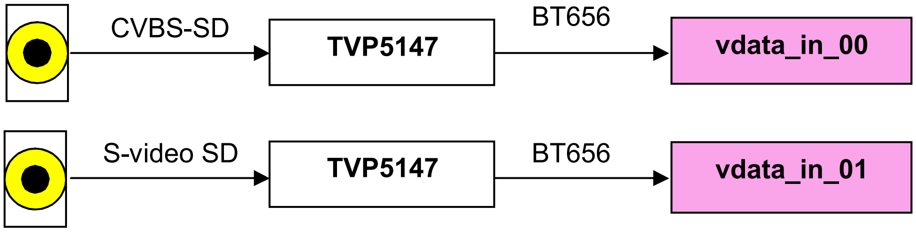


Figure 2. Physical input interface for SD on EVM

The following encoders are used for various types of Display:

- Single ADV7343 for SD channel. Encoder is connected to both S-video input and composite input which provides BT.656 input to channel2.

The following figure shows the physical connections for ADV7343 encoders on EVM.

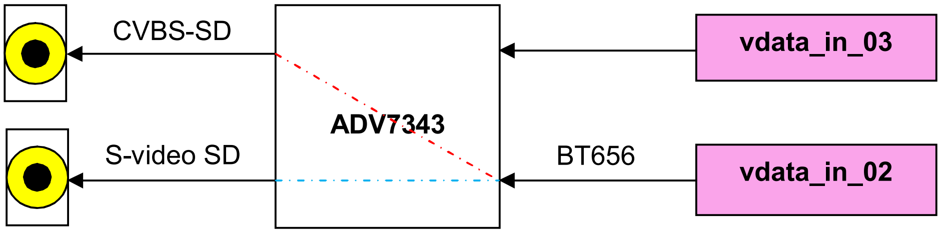


Figure 3. Physical output interface for SD on EVM

1.4.2 Software Overview

This device driver is written in conformance to the DSP/BIOS™ GIO model and handles communication to and from the VPIF device. VPIF has its own internal DMA for data handling.

The block diagram below shows the overall system architecture:

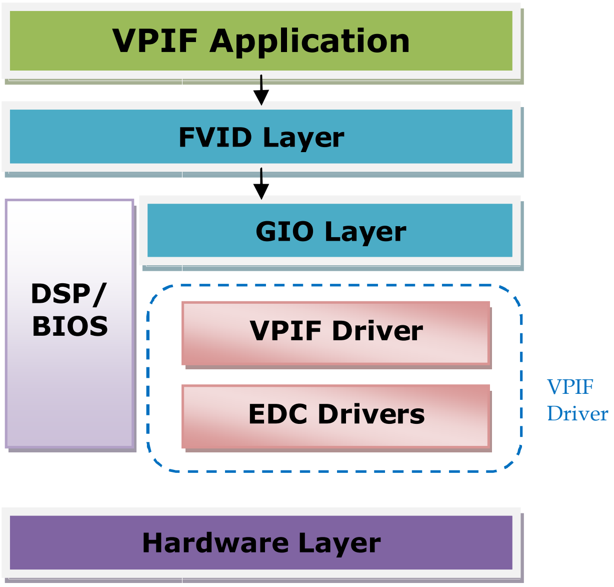


Figure 4. Overall Design

VPIF driver lies below the FVID and GIO layer. The driver uses the DSP BIOS™ APIs for OS services. The main function of the VPIF driver is to program the peripheral for the display or capture configuration and move the video data to and from SDRAM to the VPIF interface. The VPIF driver captures the video data from the decoder and outputs the video data to the encoder for display. The VPIF channel data format is selectable based on the settings of the specific Channel Control Register (Channels 0-3). The EDC drivers are used to configure the encoders and decoders using codec interface. The call to EDC drivers is always through the VPIF layer.

All channels can be activated simultaneously for SD mode

- Channels 0 and 1 are prepared only for capture.

- Channels 2 and 3 are prepared only for display.

Display applications can access VPIF channel-2 and VPIF channel-3 through software interfaces. Both the channels support SD display but only channel-2 supports HD display. Using EDC interface encoder is configured. Display Driver supports the following standards:

- SD output display: NTSC 480i 30 fps and PAL 576i 25 fps.

Capture applications can access VPIF channel-0 and VPIF channel-1 through software interfaces. Both the channels support SD capture but only channel-0 supports HD and RAW capture. Channel 0 and 1 are used simultaneously for raw video capture using sensor device Using EDC interface decoder and sensor is configured. Capture Driver supports the following standards:

- Raw input capture
- SD input capture: NTSC 480i 30 fps and PAL 576i 25 fps

⚐ *This driver is not tested for HD because of Hardware constraints. However the driver is designed keeping HD in mind.*

2 Requirements

Following are some of the requirements of VPIF driver:

1. The Driver shall conform to IOM Model of DSP/BIOS™ Operating System.
2. For field mode, each IO request to the driver would require both fields' data of a frame.
3. Supports dynamic switching among input interfaces and various resolutions with some necessary restrictions wherever applicable.

Capture Driver

4. The driver will expose 2 software channels of capture for each of the hardware channels 0 and 1. Both the software channels will support SD (BT656) mode but only channel 0 will support RAW capture.
5. The decoder EDC driver will support runtime change of the following parameters:

   TVP5147: SD BRIGHTNESS, SD CONTRAST, SD SATURATION, SD HUE and SD AUTOGAIN
6. VBI capture in the slice mode will be provided for closed caption, WSS and CGMS. It will be captured using decoder TVP5147 available on EVM.

Display Driver

7. The driver will expose 2 software channels of Display for each of the display hardware channels 2 and 3. Both the software channels will support SD (BT656) mode.
8. The encoder EDC driver will support runtime change of the following parameters:

   ADV7343: SD BRIGHTNESS, SD HUE, and SD GAMMA.
9. VBI capture in the slice mode will be provided for only closed caption, WSS and CGMS. Encoder ADV7343, available on EVM, will be used for this purpose.

2.1 Assumptions

Following are the assumptions of the driver:

1. The SD capture/display channel will support the following resolutions for BT stream:
   - NTSC 480i at 30fps
   - PAL 576i at 25fps
2. Raw Ancillary data capture/display is supported by VPIF driver provided the same is supported by encoder and decoder.
3. The VPIF driver will not allocate frame buffers for driver operations. Applications have to create buffers for this purpose. The API's for buffer allocation will be provided. It is suggested that applications should use the APIs provided with driver for frame buffer allocation /de-allocation purpose.
4. Minimum two buffers are required to be queued inside the VPIF driver before the driver is ready to start capture or display operation.

5. For capture, the driver completes the IO request once a frame is captured or both the fields are captured. For display, the driver completes the IO request once a frame is displayed or both the fields are displayed.
6. Capture driver
    a. Always returns the most recent frame.
    b. Cycle through available buffers when application falls behind.
7. Display driver
    a. Queues buffers for displaying from application.
    b. Keep displaying the same frame when running out of buffers.
    c. Returns the IO request/buffer immediately after displaying the content of that IO request, if an IO request is pending.

2.2 Constraints

Following are the constraints of the VPIF driver:

1. This driver is not tested for HD because of hardware constraints.
2. Simultaneous RAW and SD capture would not be supported by the driver.
3. Raw Capture will be supported provided EVM has support for same i.e. there should be decoder to capture RAW data.
4. HD and RAW display will not be supported.
5. As SD mode is supported by VPIF, only SD parameters are configured in the encoder and decoder.
6. Dynamic switching of resolution and dynamic switching of interfaces is not supported when streaming is on.
7. VPIF input/output buffer addresses must be multiple of eight.
8. FVID_EXCHANGE mechanism should be used for exchanging pointers between buffers.
9. Raw VBI and raw HBI is supported by the driver but not tested.
10. VBI data capture for RAW is not supported as it is not supported by VPIF.

3 Design Description

This section gives the detailed architecture of DSP/BIOS™ VPIF device driver, including the device driver partitioning, as well as deployment considerations.

The application would invoke the driver routines through the FVID Wrapper Calls, which is an adaptation layer. FVID Wrapper internally calls GIO APIs. Device drivers are accessed by the applications for performing I/O using BIOS through the above mentioned layers.

IOM is the component that performs the device specific operations. IOM Mini Driver directly controls VPIF and indirectly controls external devices like encoders and decoders through EDC driver.

The capture and display sections of the VPIF can be configured to operate independent of each other. Channels 0 and 1 of VPIF can be can be assigned to perform capture operation. Channels 2 and 3 of VPIF can be can be assigned to perform display operation.

The detailed block diagram below shows the overall system architecture:

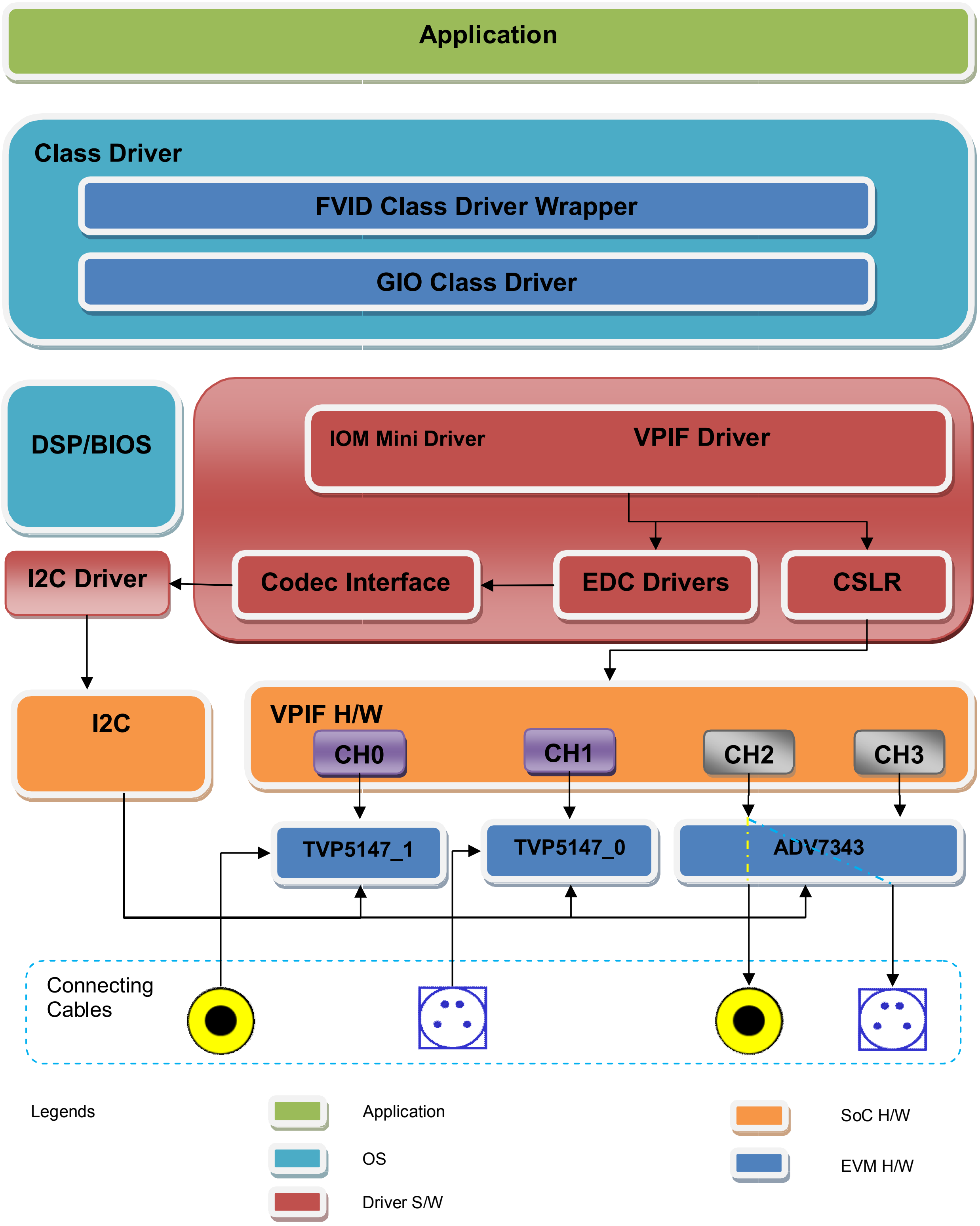


Figure 5. Detailed Design

Driver module which this document discusses lies below the GIO layer, which is a class driver layer provided by DSP BIOS™. The VPIF driver uses the rCSL (register overlay) to access the Hardware and the DSP BIOS™ APIs for OS services. The queue operation to driver is asynchronous, dequeue operation is synchronous (i.e. any dequeue operation of the driver will wait for the completion) and the exchange operation because of involving dequeue is synchronous call.

The central portion shown constitutes the mainline VPIF driver component; the surrounding modules constitute the supporting system components. The supporting system modules, do not specifically deal with VPIF, but assist the driver by providing utility services.

For more information about the IOM device driver model, see the DSP/BIOS™ Device Driver Developer's Guide. The rest of the document elaborates on the design of the VPIF Device driver.

3.1 Design Philosophy

This device driver is written in conformance to the DSP/BIOS™ IOM device driver model and handles communication to and from the VPIF hardware. The driver is designed keeping a device, also called instance, and channel concept in mind.

3.1.1 The Instance Concept

The IOM model provides the concept of the Instance for the realization of the device and its communication path as a part of the driver implementation. The instance Object maintains the state of the VPIF device. The instance word usage refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall be an instance variable. For example, DMA FIFO threshold setting is instance wide variable. Each hardware instance shall map to one VPIF instance. The "instance object" will be instantiated per physical instance of the peripheral. The lifetime of the instance is between its binding using vpifMdBindDev() and its unbinding using vpifMdUnBindDev().

3.1.2 The Channel Concept

The capture and display sections of the VPIF are represented by channels in the IOM driver. The VPIF driver provides simultaneous capture or display operations on the single VPIF instance. This is supported by providing two capture channels and two display channels per instance in order to perform IO operations. C6748/OMAPL138 SoC's contains one instance of VPIF, and the driver for this needs to maintain single instance object. Each instance has four channels supported - 2 capture and 2 display, so the instance object needs to save the four channel handles. The lifetime of the channel is between its creation using vpifMdCreateChan() and its deletion using vpifMdDeleteChan().

The VPIF peripheral needs the instance to maintain its state. The channel object holds the IOM channel state during execution.

The software channels created for VPIF driver are bound with physical channel of the peripheral.

❑ *Please refer to Low Level Definitions section for channel object structure detail.*

The following figure shows the generic instance-channel-hardware mapping for VPIF Capture/Display driver.

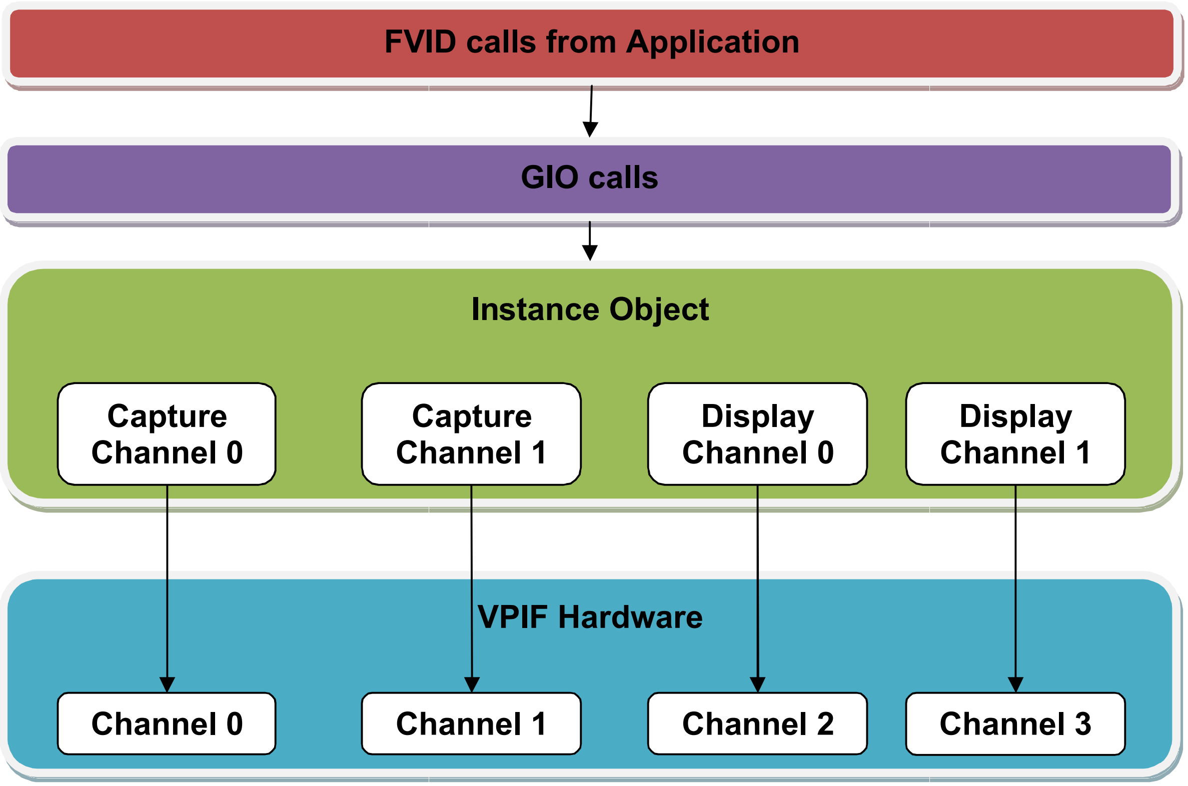


Figure 6. Instance and Channel Object

3.2 Static View

The VPIF driver is a single layer IOM compliant implementation of the device driver and coupled tightly with the DSP/BIOS™ operating system. The other two components used by the VPIF driver to function are – the HW specific bottom-edge and the EDC drivers for programming the encoders and decoders. Since there is a clear separation of roles and responsibilities, the prescribed architecture helps in creation of robust device drivers through tested/reusable pieces.

The VPIF driver contains the following important files.

VPIF Public Header File (Vpif.h)

This file specifies the list of configurable items which could be configured by the application during instantiation. It contains the following: the data structures, enums, constants, IOCTLS, and config variables that shall be exposed for the user. This header file shall be included by the applications, for referring to any of the driver data structures/components.

VPIF Local Header File (Vpif_Local.h)

The Instance Object (the driver object) and channel objects which contain all the info related to that particular IO channel are defined here. This information might be irrelevant to the User. The instance object is the container for all driver variables, channel objects etc. In essence, it contains the present state of the instance being used by the application.

VPIF Source File (Vpif.c)

The `Vpif.c` source file contains the IOM API implementations and the Ioctls supported for the VPIF driver. This file implements the core of VPIF driver and is the place where static instantiations and references to module usage are handled.

This section gives detail on the static view explaining the functional decomposition of the VPIF driver. In the following sub-sections, each of these functional sub-components and the interfaces implemented by each of the sub-component of the VPIF device driver is further elaborated.

❑ *Refer to VPIF Device Driver User Guide documentation for complete details on APIs and data structures.*

3.2.1 IOM Interface

The Application uses the FVID APIs to make use of driver routines. The IOM constitutes the Device Driver manifest to application. This adapts the Driver to DSP/BIOS™. The user of device driver will only need bother about the FVID interface, especially the upper-edge services exposed to the Application/OS. All other interfaces discussed later in this document are more of interest to people developing/maintaining the device driver.

The driver uses an internal data structure, called channel object, to maintain its state and parameters during execution. This channel is created whenever the application calls a GIO create call to the VPIF IOM module. The channel object is held inside the Instance State of the module.

A driver which conforms to the IOM driver model exposes a well define set of interfaces

- Driver initialization function.
- IOM Function pointer table.

Hence the VPIF driver (because of the virtue of its IOM driver model compliance) exposes the following interfaces.

- Vpif_init()
- Vpif_IOMFXNS

The Vpif_init() is a startup function that needs to called by the user (application) to initialize all the data structures of the Vpif driver. This function also initializes all the instance specific information for the VPIF instance like the Base address, interrupt number etc.

❑ *The working of the Vpif driver will be affected if this function is not called by the application prior to accessing the VPIF driver APIs.*

The VPIF driver exposes IOM function pointer table which contains the various APIs provided by the VPIF driver. The IOM mini-driver implements the following API interfaces to the class driver:

IOM Interfaces	Description
vpifMdBindDev()	The vpifMdBindDev() function is called by the DSP/BIOS™ after the bios initialization The vpifMdBindDev() should typically perform the following actions. ❖ Acquire the device Handle for the specified instance of the VPIF on the SoC.

	❖ Configure the VPIF device object with the specified parameters (or default parameters, if there is no external configuration. The default parameters shall be specified). ❖ Registers the VPIF driver ISR.
vpifMdUnBindDev()	The vpifMdUnBindDev() function is called to destroy an instance of the VPIF driver. ❖ It will unroll all the changes done during the bind operation and free all the resources allocated to the VPIF.
vpifMdCreateChan()	The vpifMdCreateChan() function is executed in response to the GIO_create() API call by the application. Application has to specify the mode in which the channel has to be created through the "mode" parameter. The VPIF driver supports only two modes of channel creation (input and output). ❖ It creates a communication channel in specified mode to communicate data between the application and VPIF device instance. ❖ It also configures VPIF channel and allocates required resources.
vpifMdDeleteChan()	The vpifMdDeleteChan() is invoked in response to the GIO_delete() API call by the application. ❖ It frees the channel and all the resources allocated during the creation of the channel.
vpifMdSubmitChan()	Submit an I/O packet to a channel for processing. Used for data transfer operations like enqueue, dequeue and exchange operations.
vpifMdControlChan()	Implements the IOCTLS for VPIF IOM mini driver. All control operations go through this interface.

The Application would invoke the driver routines through the FVID Wrapper Calls. FVID Wrapper internally calls GIO APIs. EDC Driver is used to configure external Video Decoder and Encoder. The EDC function table, passed as a part of channel creation, is used to open and configure the corresponding EDC driver. VPIF driver library calls EDC Driver APIs for external Decoder and Encoder configurations.

❑ *VPIF driver does not have any default configuration support. Before using the driver, application should configure the driver with valid configurations during channel creation. This is not applicable for EDC, as EDC supports default configuration.*

3.2.2 Driver Naming

The VPIF driver identifies the channel no and the encoder or decoder associated with that channel by the name argument of FVID_create(). An example of the string passed as parameter during the create call is "*/VPIF0/2/I2C0/ADV7343/0x10*". This string tells the driver to use VPIF instance 0, open channel number 2, the encoder is connected with I2C0 and the name is ADV7343 with slave address as 0x10.

3.2.3 CSLR Interface

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the VPIF driver module to configure VPIF registers. CSLR is implemented as a header file that has CSLR macros and register overlay structure which allows the access to the VPIF registers.

3.2.4 FVID Frame Format

The FVID frame structure used for data communication between VPIF and application is as follows:

```
/* Structure for FVID frame buffer descriptor */
typedef struct FVID_Frame_t
{
    QUE_Elem            queElement;
    /**< for queuing */
    union {
        FVID_IFrame     iFrm;
        /**< y/c frame buffer for interlaced mode */
        FVID_PFrame     pFrm;
        /**< y/c frame buffer for progressive mode */
        FVID_RawIFrame  riFrm;
        /**< raw frame buffer for interlaced mode */
        FVID_RawPFrame  rpFrm;
        /**< raw frame buffer for progressive mode */
        Ptr             frameBufferPtr;
        FVID_SpFrame    spFrm;
        /**< y/c frame buffer for semi planar data */
    } frame;
    Uint32              timeStamp;
    /**< Time Stamp for captured or displayed frame */
    Uint32              pitch;
    /**< Pitch parameters for given plane */
    Uint32              lines;
    /**< Number of lines per frame */
    FVID_bitsPerPixel   bpp;
    /**< Number of bits per pixel */
    FVID_colorFormat    frameFormat;
    /**< Frame Color Format */
    FVID_storageFormat  storeFormat;
    /**< Storage Format */
    FVID_VbiFrame       vbiFrm;
    /**< VBI frame */
    FVID_vbiService     vbiService;
    /**< VBI Service */
    Ptr                 userParams;
    /**< In/Out Additional User Parameters per frame */
    Ptr                 misc;
    /**< For future use */
}FVID_Frame;   /**< Typedef for structure FVID_Frame_t */
```

The format parameters are pitch, lines, bpp, frameFormat, storeFormat, and so on. These parameters are updated as per the standard selected on the channel. Please refer to the user guide for detailed explanation of the parameters.

VPIF uses semi-planer data for video. The VPIF frame format structure for video data storage is as follows:

```
/* Structure for VPIF(Semi planar) Frame */
typedef struct FVID_SpFrame_t
{
    Uint8 *y1;
    /**< Pointer for top field Y data */
    Uint8 *c1;
    /**< Pointer for top field CB/CR data */
    Uint8 *y2;
    /**< Pointer for bottom field Y data. Not used for
progressive format. */
    Uint8 *c2;
    /**< Pointer for bottom field CB/CR data. Not used for
progressive format. */
}FVID_SpFrame;
```

For interlaced *y1*, *y2*, *c1*, and *c2* are valid but for progressive only *y1*, and *c1* are valid. The chrominance data is CbCr packed.

The VPIF VBI frame format structure for vbi data storage is as follows:

```
/* Structure for VBI Frame */
typedef struct FVID_VbiFrame_t
```

```
{

    Uint8 *h1;

    /**< Pointer for top field RAW HANC data. Not used if
RAW HANC data is not required */

    Uint8 *h2;

    /**< Pointer for bottom field RAW HANC data. Not used
if RAW HANC data is not required */

    Uint8 *v1;

    /**< Pointer for top field RAW VANC data. Not used if
RAW VANC data is not required */

    Uint8 *v2;

    /**< Pointer for bottom field RAW VANC data. Not used
if RAW VANC data is not required */

    FVID_SliceFrame     *s1;

    /**< Slice VBI data structure for top field*/

    FVID_SliceFrame     *s2;

    /**< Slice VBI data structure for bottom field*/

}FVID_VbiFrame;;
```

3.2.5 FVID Frame and VPIF Buffer Allocation

Frame buffers containing video data are allocated by the application and it has entire control over these buffers. Application has to create buffers and queue into the driver and then only display or capture operation can be started. Driver shall not be allocating any frame buffer on its own. Driver however shall be validating the frame buffers passed by the application prior to queuing. Minimum three frame buffers should be queued inside the driver before starting capture or display operation.

Frame buffers are exchanged among the application and the drivers by using the FVID_dequeue(), FVID_queue() and FVID_exchange() functions. The buffer management strategies, however, are different in the capture and display drivers and are explained in a different section in this document.

Following are the two memory allocation modes:

- User allocated buffer using direct OS APIs like MEM_calloc() and MEM_free()
- User allocated buffer using APIs provided by driver like FVID_allocBuffer() and FVID_freeBuffer().

In both cases application has the responsibility of allocation of frame buffers.

If application wants to allocate frame buffer, then application can directly queue the buffers inside the driver and is not required to allocate the frame buffers from VPIF driver. **Driver validates the provided frame buffer along with the other members.** For example if a channel is opened for RAW IO mode and the *frameFormat* contains *FVID_YCbCr422_CbCrPACKED* instead of *FVID_RAW_FORMAT*, assert will happen. Some checks are due to hardware restrictions on the VPIF module.

If application wants VPIF driver to allocate the buffers, then the driver can allocate using the segment id (provided to the driver during channel creation). If segment id is 0 then the buffers are allocated from the default system heap else they are allocated from the heap define by the segment id. FVID_allocBuffer() is used to allocate the memory from the VPIF driver and FVID_freeBuffer() is used to free it. The driver allocates the memory for based on the video mode for which the channel is created. Here the other parameters are update by the driver and application can use them.

Before allocation, drivers calculate the size of each buffer based on the channel configuration parameters. For example, the size of a buffer that can hold an entire NTSC video frame is 720x480x2. The memory requirements of the buffers storing the Video data are as follow:

For Video NTSC mode i.e. 480i – 720*480*2 bytes are required by each Frame.

For Video PAL mode i.e. 576i – 720*480*2 bytes are required by each Frame.

Similarly the buffer calculations are done for RAW VBI, RAW HBI, Slice VBI and RAW capture.

⚐ *The driver does not allocate any memory on its own. Applications have to create buffers for this purpose. It is suggested that applications should use the APIs provided with driver for frame buffer allocation purpose.*

⚐ *Please note that three buffers are required to be queued inside the driver prior to start of VPIF operation.*

⚐ *For RAW capture in 10 bit mode, the buffer size will be twice because of padding done by the VPIF peripherals.*

3.2.6 VPIF SDRAM Storage Format

The different ways the buffer can be storage formats that the driver supports are:

- Filed mode storage - *Vpif_SdramStorage_FIELD*
- Frame mode storage - *Vpif_SdramStorage_FRAME*

Vpif_SdramStorage_FRAME is the frame format and *Vpif_SdramStorage_FIELD* and *Vpif_SdramStorage_FRAME* are field formats. In case of *Vpif_SdramStorage_FRAME*, buffer contains line interleaved top and bottom field data. In the *Vpif_SdramStorage_FIELD*, top and bottom field data is stored separately in the buffer. The following figures show field and frame mode storages:

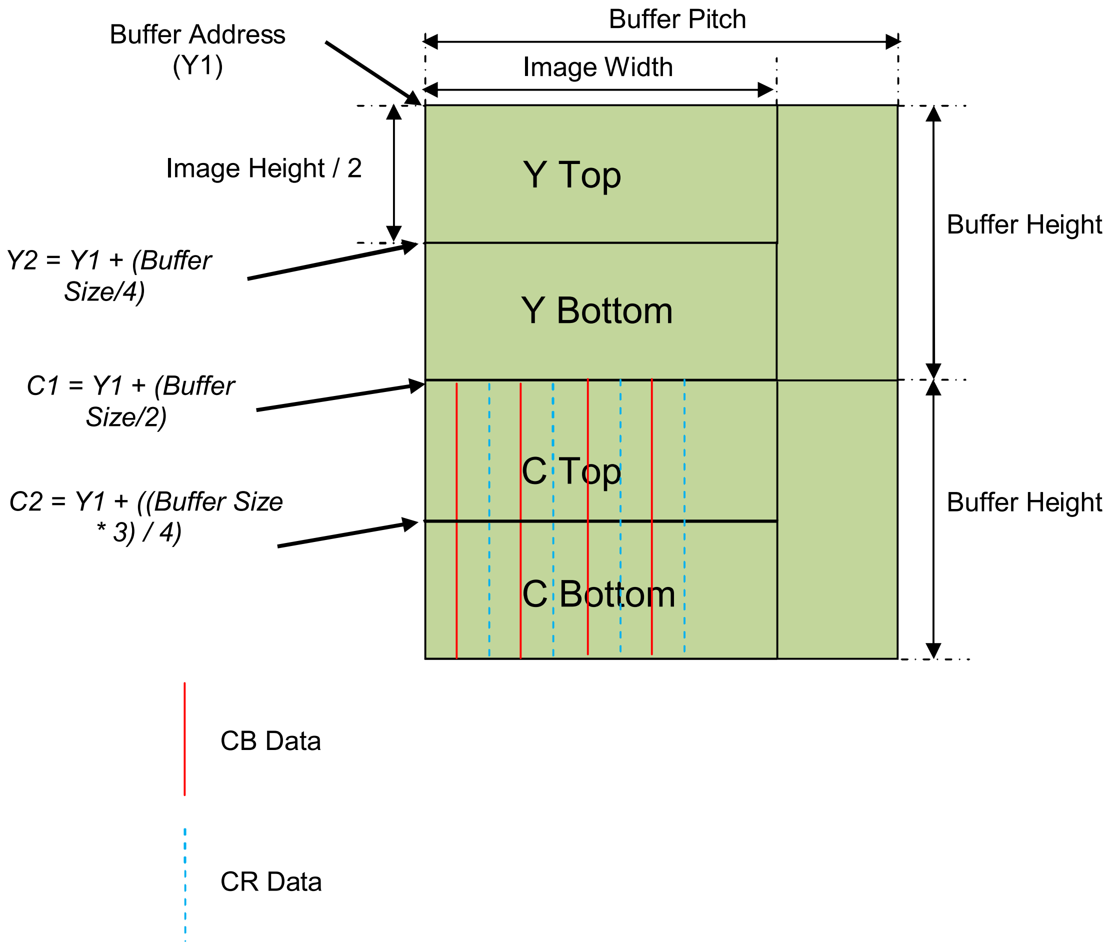


Figure 7. Field Mode Storage

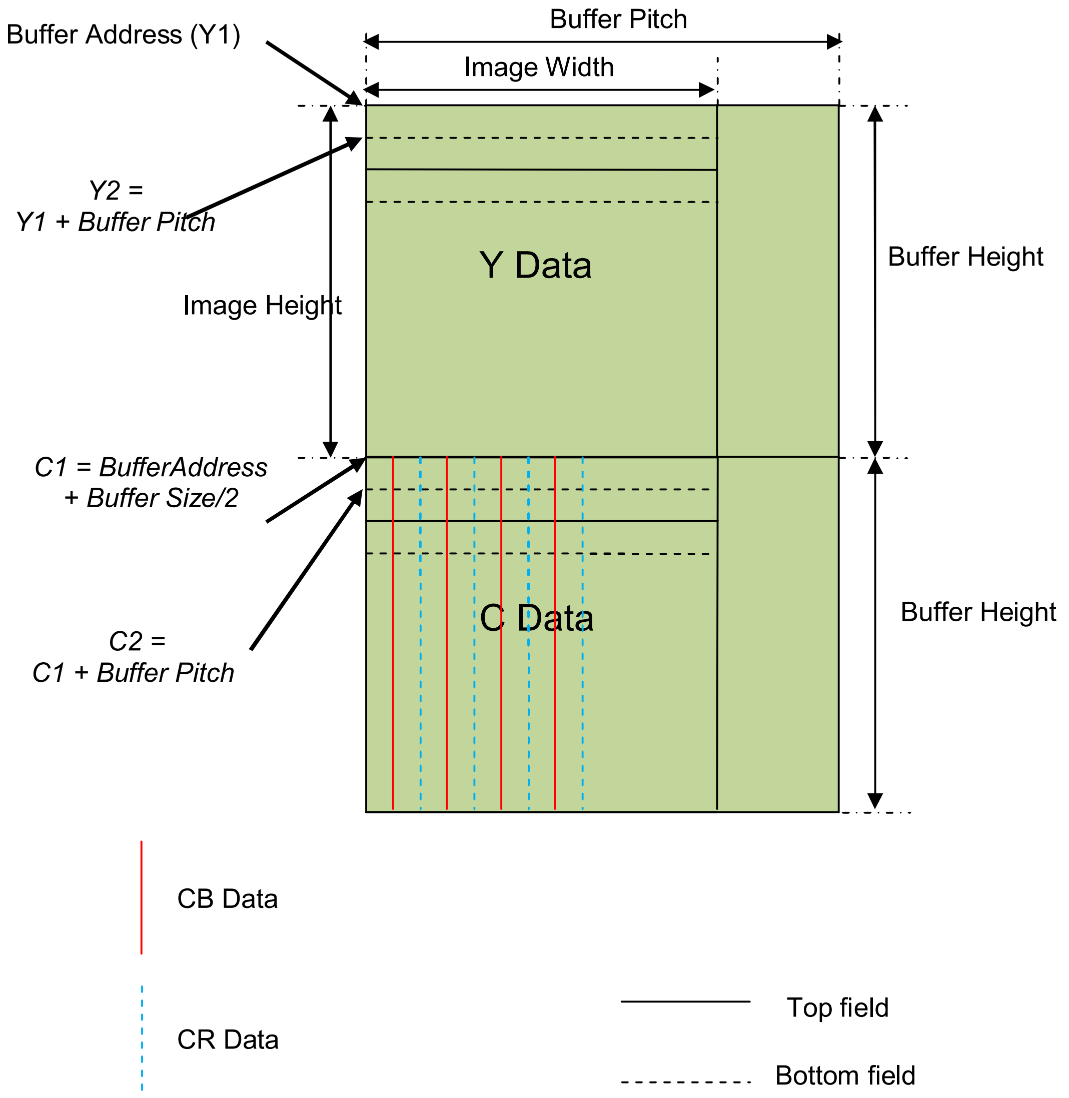


Figure 8. Frame mode storage

3.2.7 Buffer Management Strategy

- Capture driver - Always returns the most recent frame captured and cycle through available buffers when application falls behind.
- Display driver - Queues buffers for displaying from application and keep displaying the same frame when running out of buffers. Display driver returns the IO request/buffer immediately after displaying the content of that IO request, if an IO request is pending.

3.3 Dynamic View

This section gives details, explaining the deployment scenario of the VPIF driver.

3.3.1 The Execution Threads

The device driver typically implement synchronous interface to the user. The VPIF device driver operation involves following execution threads:

- BIOS thread: Function to load VPIF driver will be under BIOS OS initialization.
- Application thread: Creation of channel, Control of channel, deletion of channel and processing of VPIF data will be under application thread. All synchronous IO occur in the application thread of control, the calling thread may suspend for the requested transaction to complete.

3.3.2 Capture/Display using VPIF driver

The Vpif_init() function of the IOM layer is invoked first and is responsible for initializing the device object and channel object structure of the VPIF IOM driver. After that the driver is created by the vpifMdBindDev() of mini driver of VPIF driver.

A channel instance handle is obtained by the application by a call to FVID_create() API (corresponding GIO call is GIO_create()). The channel handle represents a unique communication path between the application and VPIF device driver. All subsequent operations that communicate to the driver shall use this channel handle. A channel object typically maintains data fields related to a channel's mode, I/O request queues, driver state information, etc. Application should relinquish channel resources by deleting all channel instances when they are no longer needed through a call to FVID_delete() (corresponding GIO call is GIO_delete()).

The application calls FVID_allocBuffer() and FVID_freeBuffer() (corresponding GIO call is GIO_control()) to allocate and free the frame buffer. Application can perform IO operation using FVID_exchange(), FVID_queue() and FVID_dequeue() calls (corresponding GIO call is GIO_submit() and corresponding IOM function is vpifMdSubmitChan()). The buffers to be captured or displayed should be passed as argument to these API's. Driver will return the status of buffer exchanged for capture or display operation e.g. *IOM_COMPLETED* for success. FVID_queue is used to relinquish a video buffer back to the driver. FVID_dequeue is used to get a pointer for allocated buffer from driver to application.

When a mini-driver completes its processing, usually in an ISR context, it calls its registered callback function to pass the IO packet back to the device independent layer of the VPIF driver and the device independent layer of the driver in turn calls the application specified callback for that particular I/O request.

Below table provides list of APIs supported by FVID driver:

Function	Description
FVID_create	Initialize the VPIF channel object
FVID_delete	De-allocate an FVID channel object
FVID_control	Send device-specific control command to the mini-driver
FVID_exchange	Exchange a frame buffer with application for a frame buffer with the driver.
FVID_dequeue	Get a video buffer from driver to application.
FVID_queue	Relinquish a video buffer back to the driver.
FVID_allocBuffer	Allocate a frame buffer using the driver's memory allocation routines.
FVID_freeBuffer	Free the buffer allocated via FVID_allocBuffer().

3.3.3 Functional Decomposition

Following are the functions at IOM layer for Capture/Display driver:

3.3.3.1 Driver Creation (Driver Initialization and Binding)

The VPIF IOM driver initializes the global data used by the VPIF driver. The initialization function for the VPIF driver is not included in the *IOM_Fxns* table, which is exported by the VPIF driver; instead a separate extern is created for use by the DSP/BIOS™. The function also sets the "*inUse*" field of the Vpif instance module object to FALSE so that the instance can be used by an application which will create it.

The binding function (vpifMdBindDev()) of the VPIF IOM mini-driver is called while creation of the driver. User is expected to invoke vpifMdBindDev(), way up in the application startup phase, perhaps in a central driver initialization function. Each driver instance corresponds to one hardware instance of the VPIF. This function shall typically perform the following actions:

- Check if the instance being created is already in use by checking the Module variable "*inUse*". Check the instance state of the driver. Validated the user supplied parameters.
- Update the instance object with the use supplied parameters.
- Update CPU event numbers.
- Base address of VPIF peripheral.
- Initialize all the channel objects with default information.
- Clear Interrupt status and channel control register of VPIF.
- Register interrupt handler of the driver.
- Configure the VPIF hardware with the user supplied one time configuration field like program the DMA size.
- Set module variable "*inUse*" to TRUE.
- Set the device instance state to *Vpif_DriverState_CREATED*.
- Update device handle and return success or error.

All these operations in effect, constitute the loading of VPIF Driver.

❑ *The user provided device parameter structure will be of type "Vpif_Params". Refer to Vpif_Params in Low Level Definitions section.*

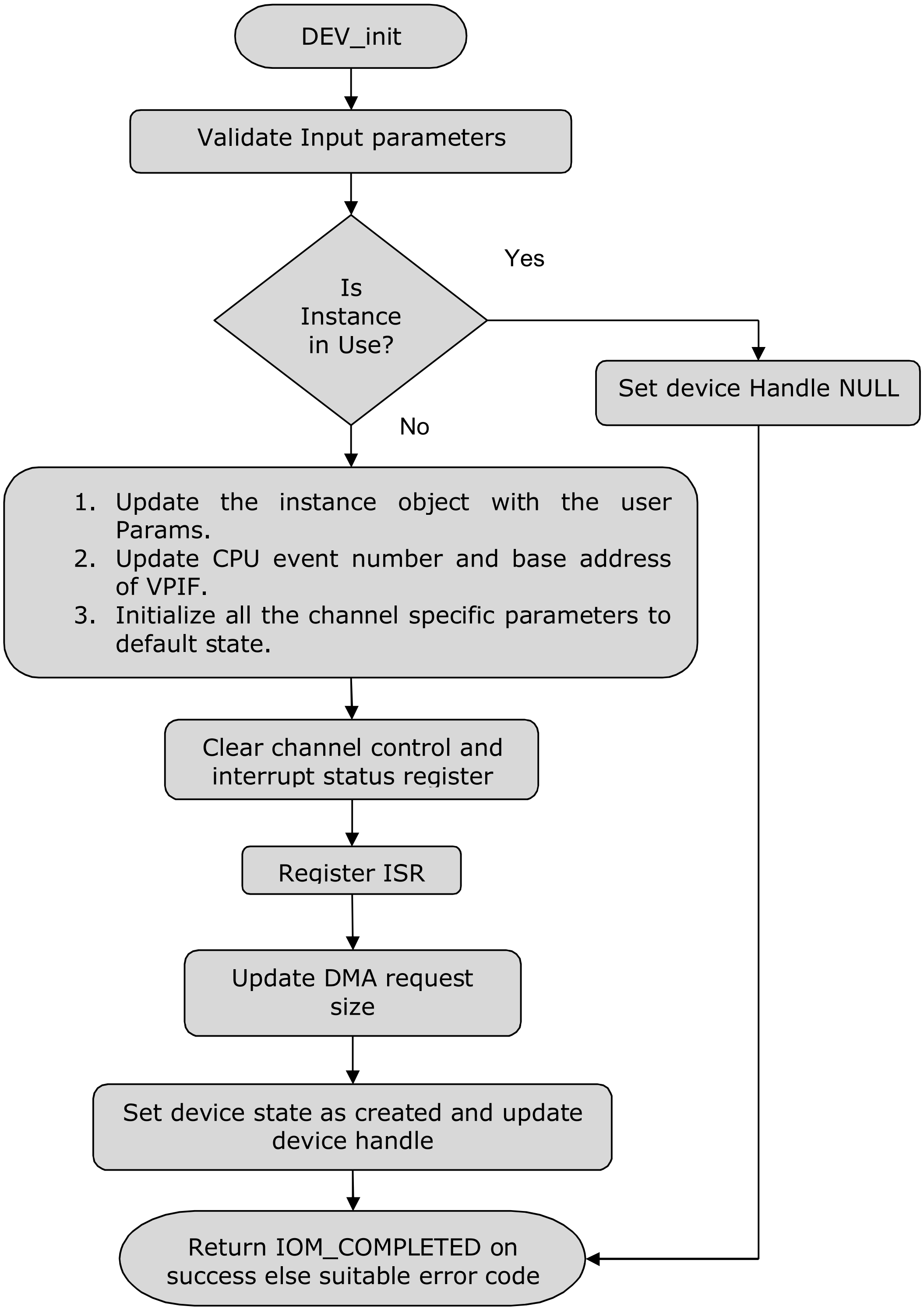

DEV_init
Validate Input parameters
Is Instance in Use?
Yes
No
Set device Handle NULL
1. Update the instance object with the user Params.
2. Update CPU event number and base address of VPIF.
3. Initialize all the channel specific parameters to default state.
Clear channel control and interrupt status register
Register ISR
Update DMA request size
Set device state as created and update device handle
Return IOM_COMPLETED on success else suitable error code

Figure 9. Driver Instance Binding flow diagram

3.3.3.2 Channel Creation

The application once it has created the device instance, needs to create a communication channel for transactions with the underlying hardware. As such a channel is a communication interface between the driver and the application.

The VPIF IOM driver allows at most four channels to be created. They are:

1. Two channels for Capture (*`IOM_INPUT`*). The driver will expose two software channels (0 and 1), one for each hardware channels (0 and 1). Both the software channels will support SD capture but only channel 0 will support HD capture and RAW capture.
2. Two channels for Display (*`IOM_OUTPUT`*). The driver will expose two software channels (2 and 3) one each for hardware channel (2 and 3). Both the software channels will support SD capture but only channel 2 will support HD Display

⚐ *This driver is not tested for HD because of Hardware constraints. However the driver is designed keeping HD in mind.*

The application can create a communication channel by calling FVID_create() call from the application with the device name, mode of operation etc. as channel parameters. When application calls FVID_create(), corresponding GIO function GIO_create() API is called which in turn calls VPIF IO mini driver's vpifMdCreateChan() function internally.

The application shall call vpifMdCreateChan with the appropriate "*`mode`*" (*`IOM_INPUT`* or *`IOM_OUTPUT`*) parameter for the type of the channel to be created. Application has to pass proper channel configuration parameters as an argument to FVID_create().Channel created with mode *`IOM_OUTPUT`* will be used for transmission of video data (e.g. video display) whereas the channel created with mode IOM_INPUT will be used for receiving video data (e.g. video capture). The user can supply the parameters which will characterize the features of the channel. VPIF driver contains separate Capture and Display channel parameters at IOM layer although their interface is same to class driver. The user can use "*`Vpif_CapChanParams`*" structure to specify the parameters to configure the capture channel and "*`Vpif_DisChanParams`*" structure to specify the parameters to configure the display channel. If the user has provided incorrect parameters like incorrect mode name at the time of channel creation, the application will not be able to open the driver.

When the application calls the vpifMdCreateChan(), driver entry point is created. The callback is registered.

The vpifMdCreateChan() function typically does the following.

- It validates the input parameters given by the application.
- It checks if the requested channel is already opened or not. If it is already opened the driver will flag an error to the application else the requested channel will be allocated.
- It updates the channel object with the user supplied parameters and some of them to default state.
- Based up on the *`IOM_INPUT`* or *`IOM_OUTPUT`* and "*`mode`*" parameter of channel params, "*`channelMode`*" is updated
- Create QUE for operations.

- If external device handle is provided calls the open interface of the external device (encoder, decoder or sensor).
- Check if the Video standard parameters like width, height, l1, l2 etc. is given by application. If yes, copy them. If no, then update the parameters depending upon the “*mode*” (e.g. NTSC, PAL, etc.) passed in channel parameter by application by looking internal lookup table.
- If “*capVbiService*” or “*dispVbiService*” parameter is not equal to *Vpif_VbiServiceType_NONE*, driver checks whether current standard supports VBI or not. If it does not, driver returns error. Otherwise, it suggests that RAW VBI, RAW HBI or Slice VBI service is required. Update the buffer size and parameters for the same.
- Calculate the pitch and buffer size for video buffers.
- Depending on “*frameFmt*” (Progressive or Interlaced) parameter calculate the “*frameStorageMode*” and “*lineOffset*”.
- Create VBI tasklet for the channel for Slice VBI.
- If the complete process of channel creation is successful, then the application will return the status and update the channel handle. This Handle should be used by the application for further transactions with the channel. This Handle will be used by the driver to identify the channel on which the transactions are being requested.

⚐ *Application has to pass the display channel parameters (Vpif_DisChanParams) for display and capture channel parameters (Vpif_CapChanParams) for capture. Also for display app. should pass IOM_OUTPUT and for capture it should pass IOM_INPUT.*

⚐ *For RAW capture only channel 0 needs to be created and channel 1 should not be used since RAW capture requires two capture channels.*

⚐ *User can pass the video parameters using member “dispVideoParams” or “capVideoParams” of structure Vpif_DisChanParams or Vpif_CapChanParams. This is an optional parameter. If not used, set this element to NULL. If set to NULL, the driver will read the parameters depending upon the mode set. If it is not NULL, its value will prevail over whatever mode being set.* ***CAUTION: If wrong parameters are sent, the driver does not verify the validity of these parameters.***

⚐ *To add video formats to driver look up table (LUT), add them in ch0Params, ch1Params or ch2Params for channel 0, 1, 2 and 3. Note that channel 2 and 3 share same structure as they support same format of standard. vpifConfigParams variable contains all the different format supported by different channel. Also increase the max mode macros supported by the channel, when a new standard is added inside the driver.*

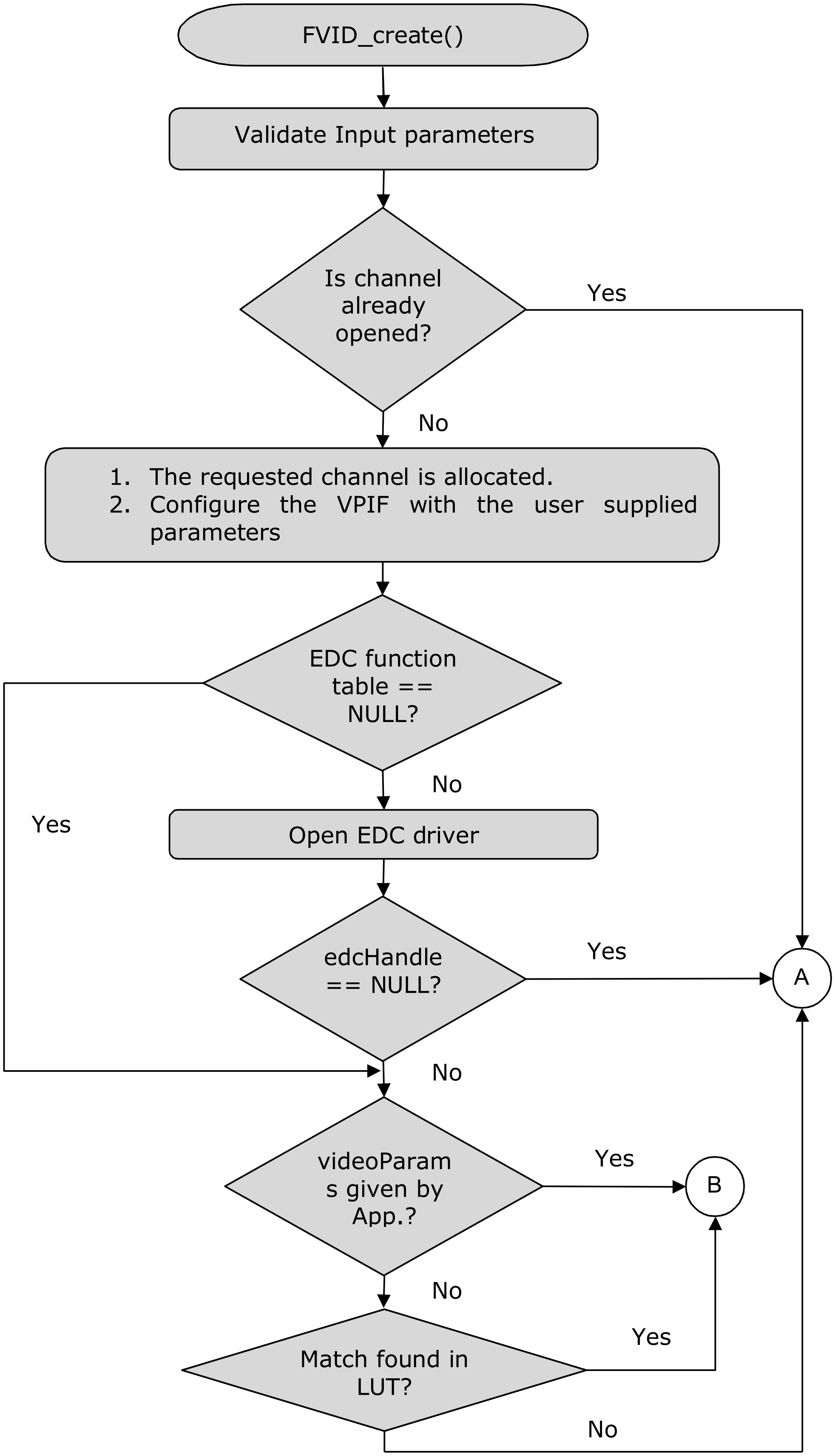
□ *The user should provide channel parameter string “name” in the following format to address the channel:*

```
“/VPIF0/0/I2C0/TVP5147_0/0x5C”

“/VPIF0” => VPIF instance 0.

“/0” => channel no 0.
```

“/I2C0” => device (I2C) with instance no., used to interface encoder/decoder.

“/TVP5147_0” => TVP instance 0 decoder is used.

“/0x5C” => I2C slave address of encoder/decoder/sensor.


FVID_create()
Validate Input parameters
Is channel already opened?
Yes
No
1. The requested channel is allocated.
2. Configure the VPIF with the user supplied parameters
EDC function table == NULL?
Yes
No
Open EDC driver
edcHandle == NULL?
Yes
A
No
videoParams given by App.?
Yes
B
No
Match found in LUT?
Yes
No

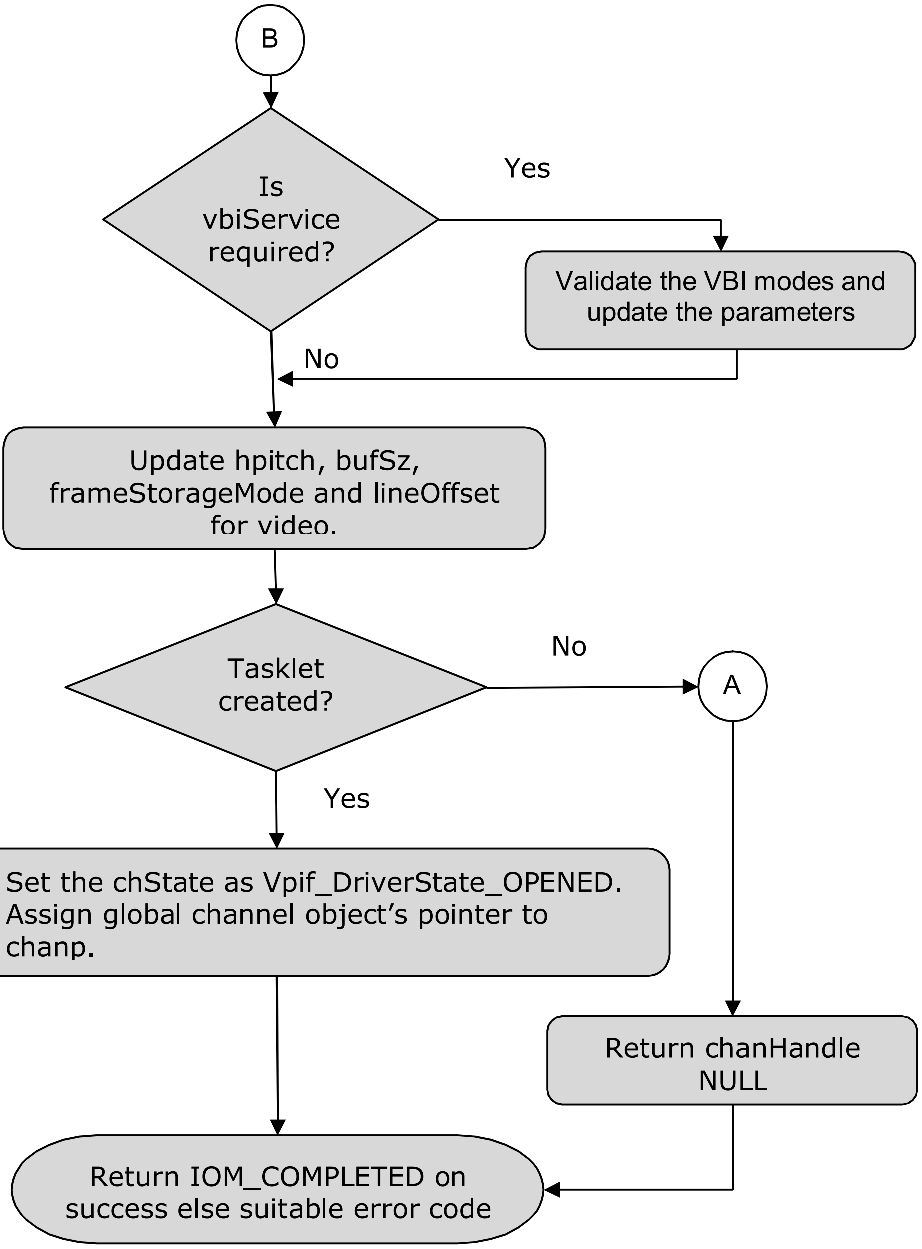

B
Is vbiService required?
Yes
Validate the VBI modes and update the parameters
No
Update hpitch, bufSz, frameStorageMode and lineOffset for video.
Tasklet created?
No
A
Yes
1. Set the chState as Vpif_DriverState_OPENED.
2. Assign global channel object's pointer to chanp.
Return chanHandle NULL
Return IOM_COMPLETED on success else suitable error code

Figure 10. Driver channel creation flow diagram

3.3.3.3 IO Access

VPIF IOM driver provides an interface to enqueue and dequeue frames to the driver. Application can call FVID_dequeue() to dequeue the buffers, FVID_exchange() to exchange the buffers and FVID_queue() to queue the buffers. When application calls them, corresponding GIO function GIO_submit() API is called which in turn calls VPIF IO mini driver's vpifMdSubmitChan() function internally. These APIs enqueues and dequeues frames containing all the parameters needed by the IOM driver.

This function handles buffer management and exchange between VPIF driver and application. The vpifMdSubmitChan() function of the VPIF IOM driver must handle command code passed to it. vpifMdSubmitChan() is called by three different FVID layer calls and handles the following commands codes:

1. FVID_dequeue(*FVID_DEQUEUE*)

   This ioctl is used to dequeue the buffer from buffer queue. This ioctl will dequeue empty display buffer or filled capture buffer from the driver and provide them to application. This ioctl is one of necessary ioctl for the streaming IO.

2. FVID_exchange(*FVID_EXCHANGE*)

   This command performs both enqueing and dequeing in one go. Once VPIF operation is started the application should call this command for exchanging buffers. For display it takes the filled data from the application and returns and empty buffer to application and vice versa for capture.

❑ *FVID_EXCHANGE mechanism should be used for exchanging pointers between buffers.*

3. FVID_queue(*FVID_QUEUE*)

   This ioctl is used to en-queue the buffers from application to the driver buffer queue. This ioctl will en-queue filled buffer to be displayed in the buffer queue of the driver. This ioctl is one of necessary ioctl for the streaming IO. If no buffer is en-queued before starting streaming, driver returns an error as there is no buffer available. So at-least two buffers must be en-queued before starting streaming. This ioctl is also used to en-queue filled buffers to be displayed after streaming is started. This ioctl should be used initially before VPIF operation is started. Application should enqueue the buffers to the driver before the start of VPIF operation.

❑ *Minimum three buffers are required to be queued inside the driver to start the operation.*

❑ *Whenever a Frame is queued inside the driver, the driver checks validity of the user specified values and asserts if they are incorrect. If they are correct, it updates the packet inside the driver queue.*

A single call to FVID_queue()/FVID_dequeue() must pass all the data like video data, RAW VBI data, RAW HBI data and sliced VBI data belonging to one video frame. That is a structure of type *FVID_VpifFrame*. To get the appropriate video data and ancillary data from this structure application needs to check the appropriate buffers pointers inside.

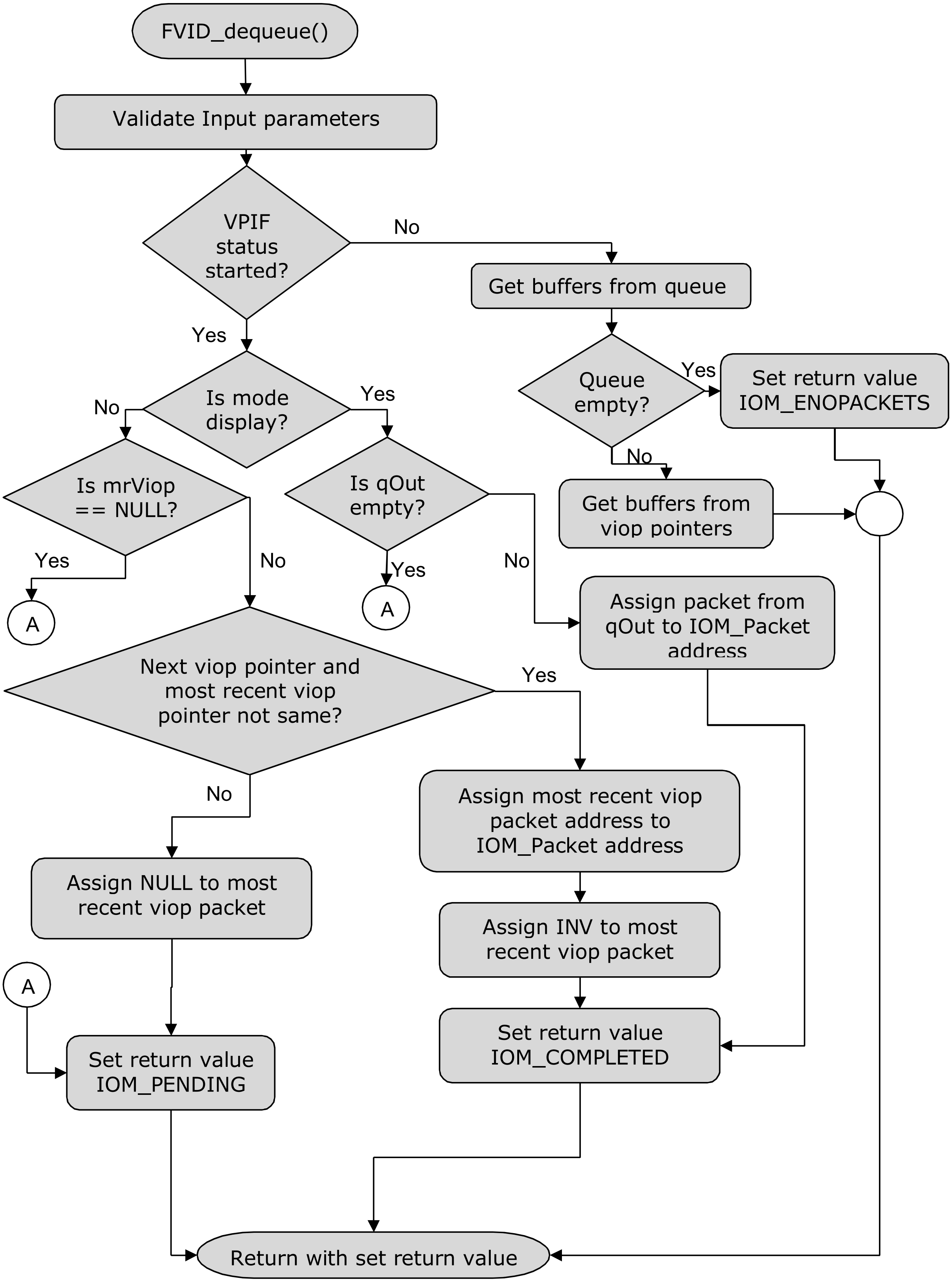

FVID_dequeue()
Validate Input parameters
VPIF status started?
No
Get buffers from queue
Yes
Is mode display?
No
Yes
Queue empty?
Yes
Set return value IOM_ENOPACKETS
No
Is mrViop == NULL?
Is qOut empty?
Get buffers from viop pointers
Yes
No
Yes
No
A
A
Assign packet from qOut to IOM_Packet address
Next viop pointer and most recent viop pointer not same?
Yes
No
Assign most recent viop packet address to IOM_Packet address
Assign NULL to most recent viop packet
Assign INV to most recent viop packet
A
Set return value IOM_COMPLETED
Set return value IOM_PENDING
Return with set return value

Figure 11. FVID_dequeue() flow diagram

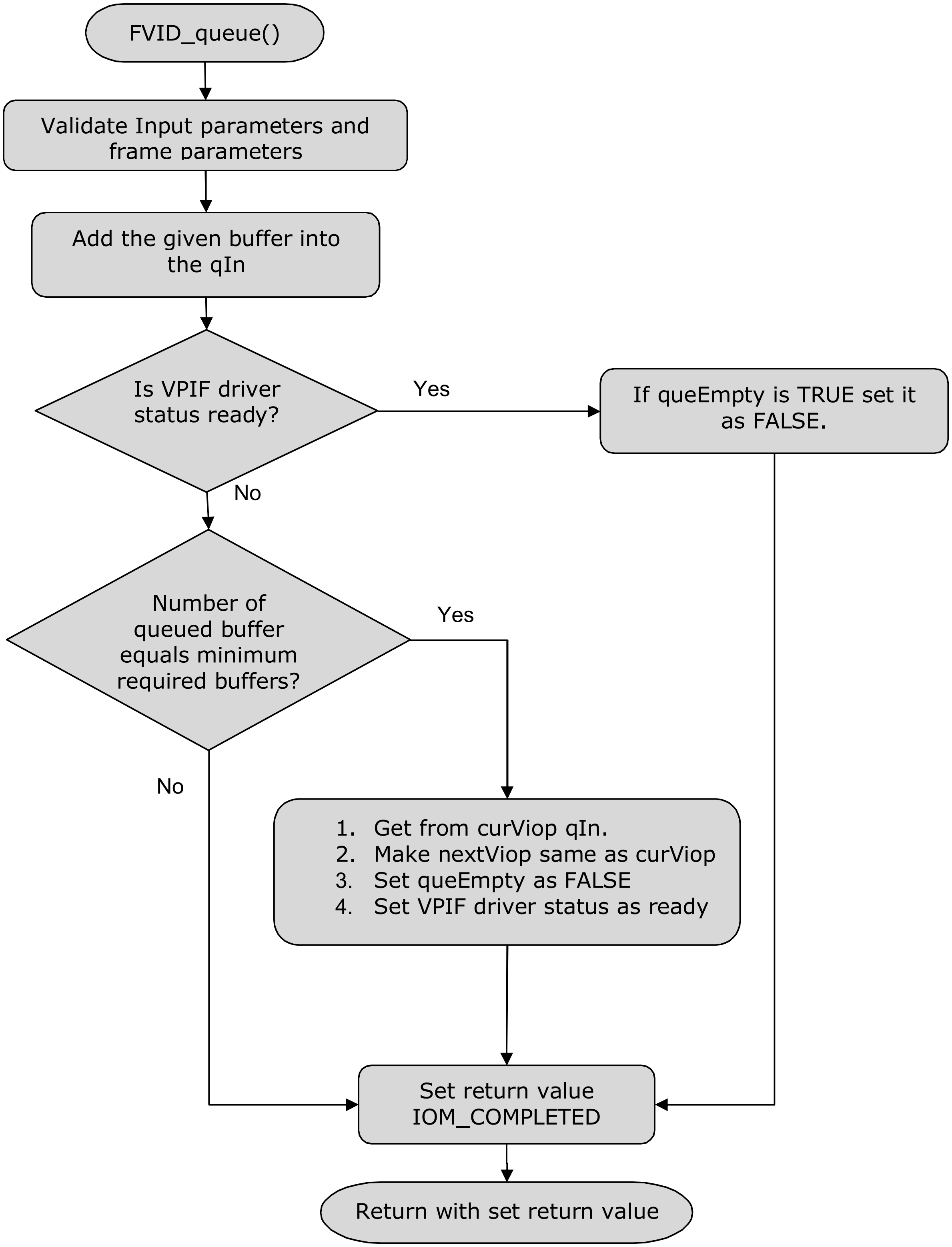

FVID_queue()
Validate Input parameters and frame parameters
Add the given buffer into the qIn
Is VPIF driver status ready?
Yes
If queEmpty is TRUE set it as FALSE.
No
Number of queued buffer equals minimum required buffers?
Yes
No
1. Get from curViop qIn.
2. Make nextViop same as curViop
3. Set queEmpty as FALSE
4. Set VPIF driver status as ready
Set return value IOM_COMPLETED
Return with set return value

Figure 12. FVID_queue() flow diagram

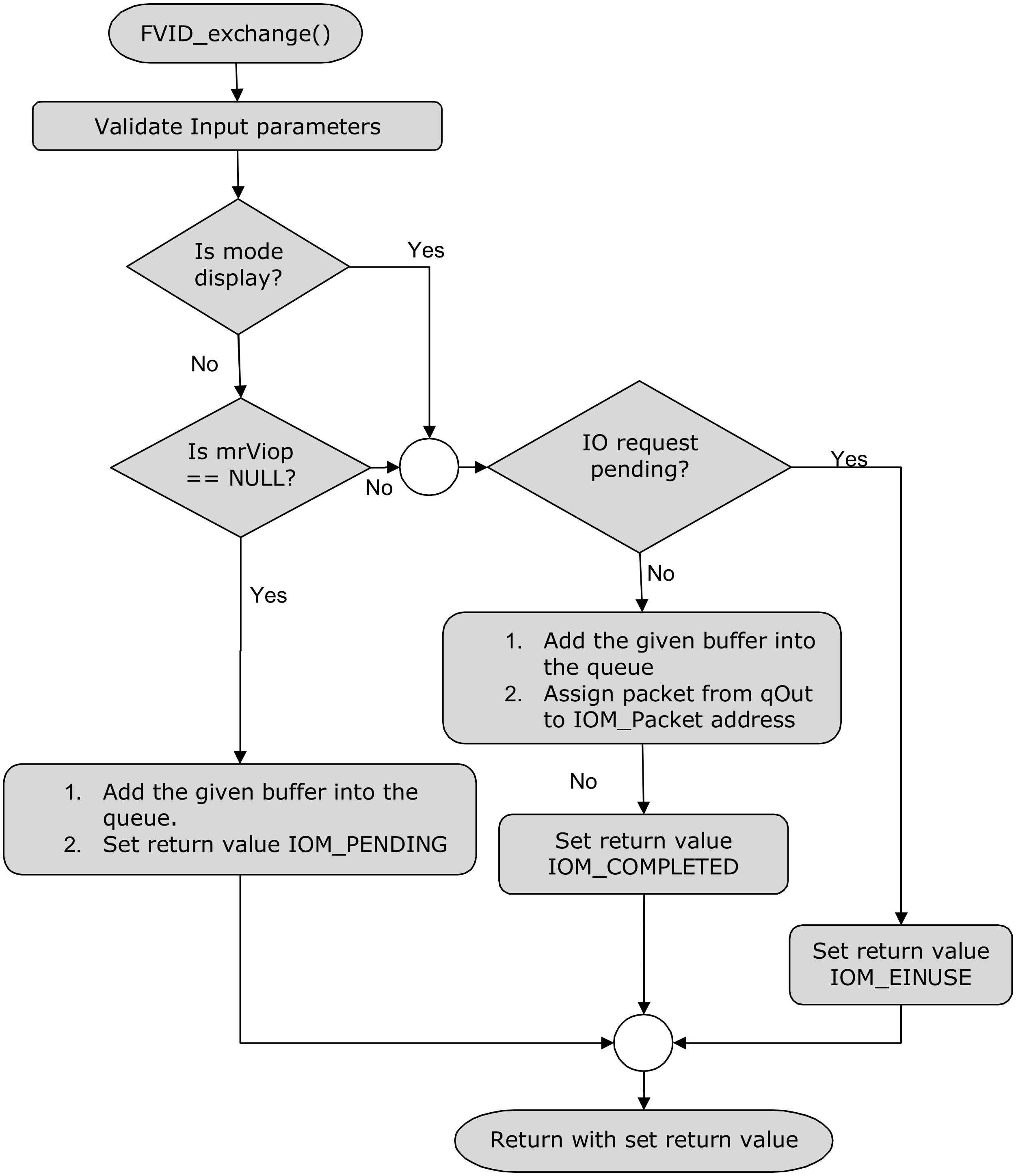

FVID_exchange()
Validate Input parameters
Is mode display?
Yes
No
Is mrViop == NULL?
No
IO request pending?
Yes
No
Yes
1. Add the given buffer into the queue
2. Assign packet from qOut to IOM_Packet address
No
1. Add the given buffer into the queue.
2. Set return value IOM_PENDING
Set return value IOM_COMPLETED
Set return value IOM_EINUSE
Return with set return value

Figure 13. FVID_exchange() flow diagram

3.3.3.4 Control Commands

VPIF IOM driver implements device specific control functionality which may be useful for any application, which uses the driver. VPIF driver provides an interface for controlling the driver by issuing a set of specified control commands. The application can use the control commands to control the functionality of VPIF driver. Application may invoke the control functionality through a call to FVID_control(), FVID_allocBuffer() and FVID_freeBuffer() corresponding GIO function GIO_control() API is called which in turn calls VPIF IO mini driver's vpifMdControlChan() function internally.

The typical control flow for the VPIF control function is as given below:

- Validate the command sent by the application.
- Check if the appropriate arguments are provided by the application for the execution of the command.
- Process the command and return the status back to the application.

❑ *External device configuration is also done by calling vpifMdControlChan() with EDC specific control commands.*

vpifMdControlChan() supports the following control functionality:

1. Vpif_IOCTL_CMD_START

   This ioctl is used to start video display or capture channel functionality. If streaming is already started, this ioctl call returns an error.

2. Vpif_IOCTL_CMD_STOP

   This ioctl is used to stop video display or capture channel functionality. If streaming is not started, this ioctl call returns an error.

3. Vpif_IOCTL_CMD_GET_NUM_IORQST_PENDING

   This ioctl returns the number of request pending in queue.

4. Vpif_IOCTL_CMD_GET_CHANNEL_STD_INFO

   This ioctl returns the standard channel information to application. The ioctl returns the information in `Vpif_StdInfo` structure.

5. Vpif_IOCTL_CMD_CHANGE_RESOLUTION

   This IOCTL will be used to change the current resolution for the VPIF channel. Application has to pass pointer to `Vpif_ConfigParams` variable as a third argument in vpifMdControlChan() function.

❑ *Please note that changing the resolution between SD, HD and RAW mode is not allowed i.e. Channel properties cannot be changed (Application may need to close the channel and create channel in that case). Using this IOCTL the application can switch between different resolutions with in SD (PAL to NTSC) or HD (720P to 1080P) or RAW (VGA to SVGA).*

❑ *Please use either of the following:*

a) *Application can choose to specify the pre-defined modes (`Vpif_VideoMode`) in the "`mode`" parameter*

b) *Application can set the "`mode`" parameter to "`Vpif_VideoMode_NONE`" and provide the filled up `Vpif_ConfigParams` structure as third parameter.*

c) *If application sets valid mode in "`mode`" parameter and also sends the filled structure, the driver would consider the "`mode`" parameter and update accordingly.*

❑ *The driver does not check the validity for these parameters when application uses point b method mentioned above.*

⚐ *The channel should be stopped using `Vpif_IOCTL_CMD_STOP` IOCTL. Also the buffers should be freed up, as the buffer requirement changes once the resolution changes.*

6. FVID_ALLOC_BUFFER

   This ioctl is used to allocate frame buffer structure. It also allocate frame buffer and assign it to frame buffer structure.

7. FVID_FREE_BUFFER

   This ioctl is used to free the frame buffer and frame buffer structure allocated previously by *`FVID_ALLOC_BUFFER`* ioctl.

8. Default: External device configuration are done by calling vpifMdControlChan() with EDC specific control commands. The command passed to the underlying EDC driver is (cmd - *`Vpif_IOCTL_CMD_MAX`*). Refer to EDC section for more detail.

The basic control flow for the handling of the control commands for the driver is shown below. Also note that some of the individual command handling is also detailed here.

❑ *Please refer to the user guide on information about the argument passed with these IOCTL's.*

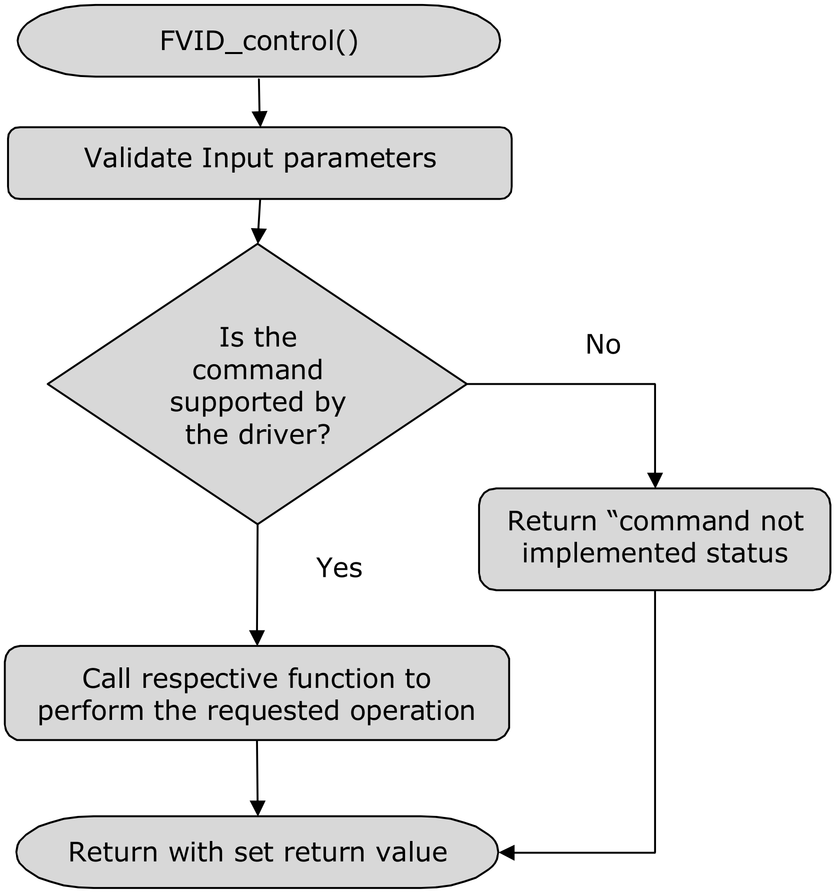
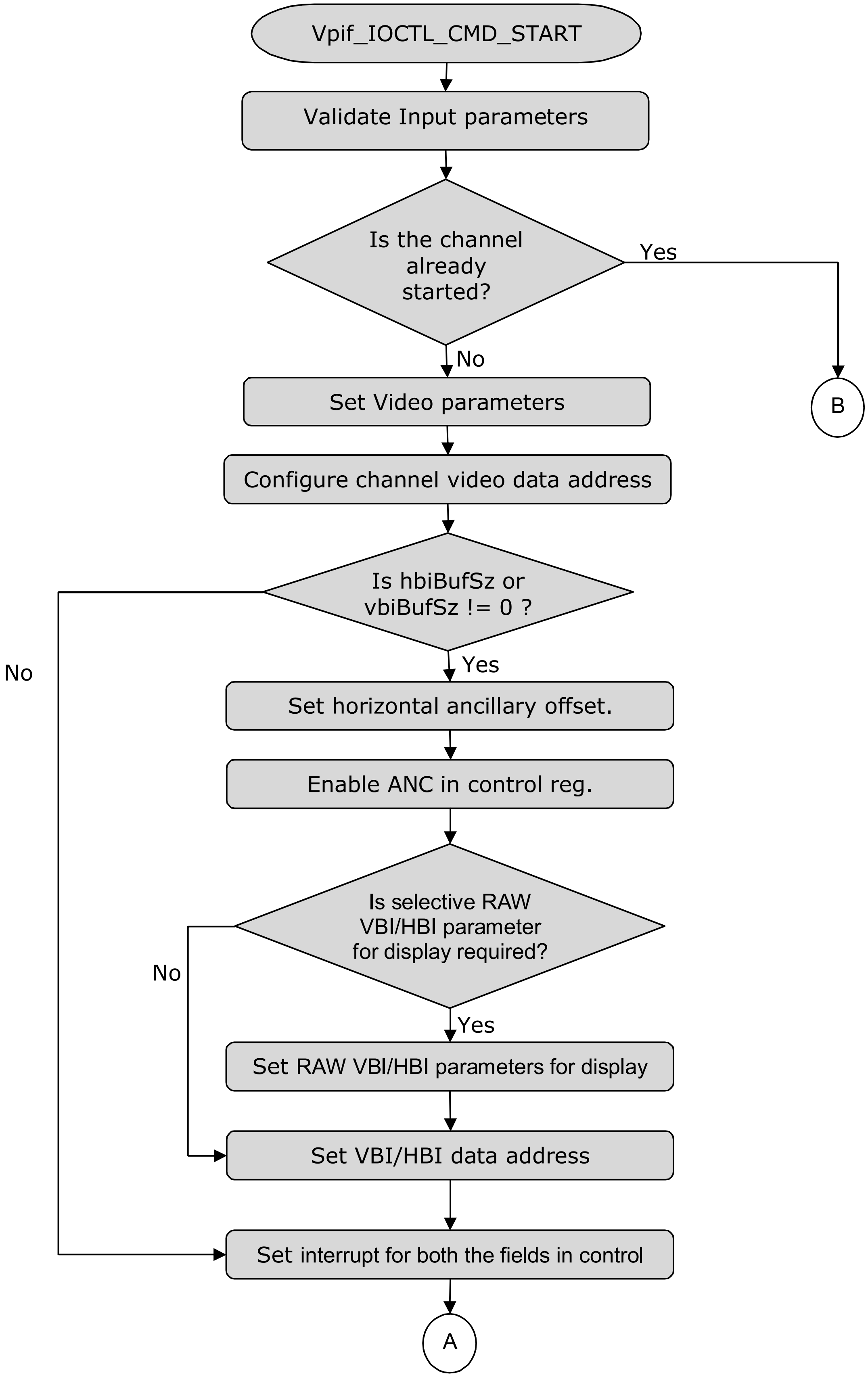

FVID_control()
Validate Input parameters
Is the command supported by the driver?
No
Return "command not implemented status
Yes
Call respective function to perform the requested operation
Return with set return value

Figure 14. Driver control command flow diagram


Vpif_IOCTL_CMD_START
Validate Input parameters
Is the channel already started?
Yes
B
No
Set Video parameters
Configure channel video data address
Is hbiBufSz or vbiBufSz != 0 ?
No
Yes
Set horizontal ancillary offset.
Enable ANC in control reg.
Is selective RAW VBI/HBI parameter for display required?
No
Yes
Set RAW VBI/HBI parameters for display
Set VBI/HBI data address
Set interrupt for both the fields in control
A

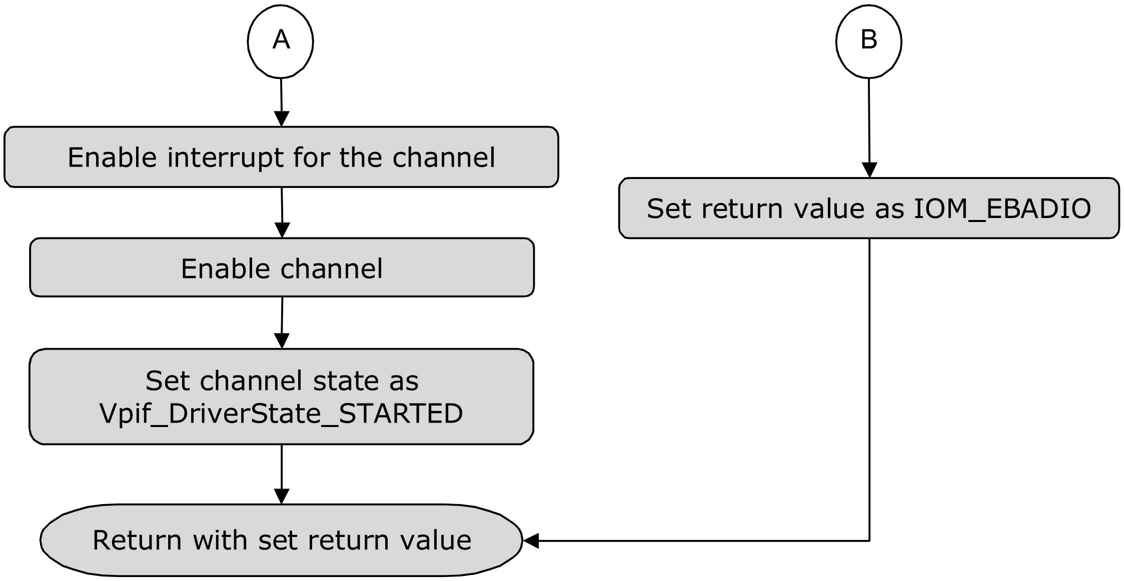

A
B
Enable interrupt for the channel
Set return value as IOM_EBADIO
Enable channel
Set channel state as Vpif_DriverState_STARTED
Return with set return value

Figure 15. Vpif_IOCTL_CMD_START flow diagram

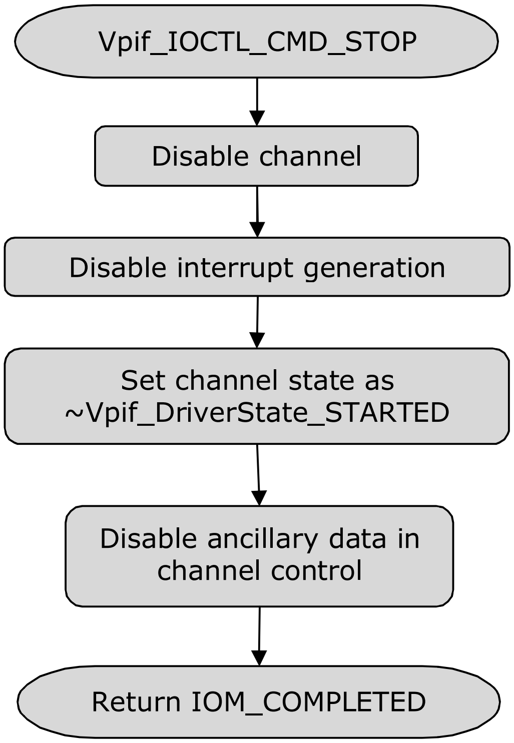

Vpif_IOCTL_CMD_STOP
Disable channel
Disable interrupt generation
Set channel state as ~Vpif_DriverState_STARTED
Disable ancillary data in channel control
Return IOM_COMPLETED

Figure 16. Vpif_IOCTL_CMD_STOP flow diagram

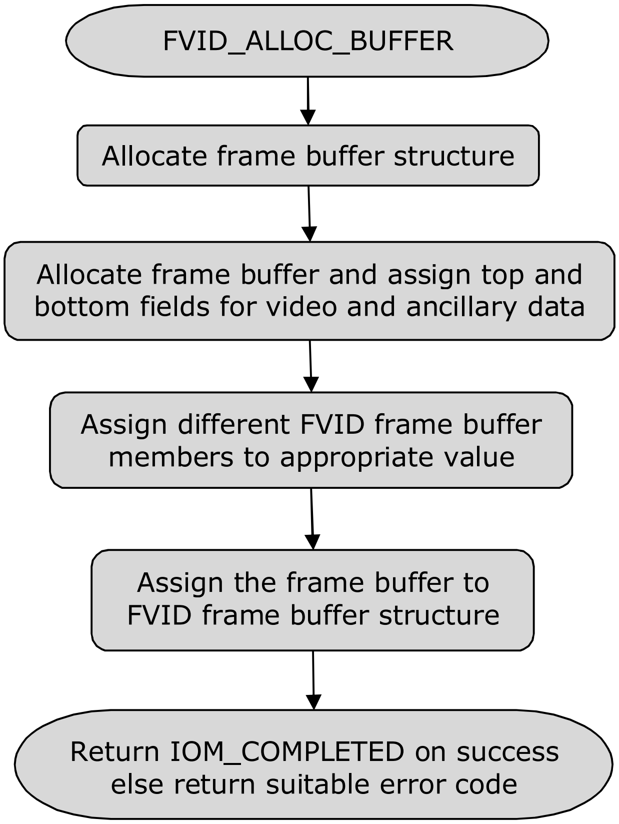

FVID_ALLOC_BUFFER
Allocate frame buffer structure
Allocate frame buffer and assign top and bottom fields for video and ancillary data
Assign different FVID frame buffer members to appropriate value
Assign the frame buffer to FVID frame buffer structure
Return IOM_COMPLETED on success else return suitable error code

Figure 17. FVID_ALLOC_BUFFER flow diagram

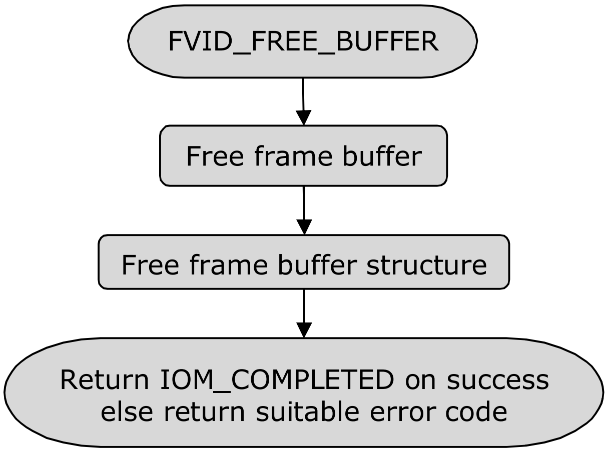

FVID_FREE_BUFFER
Free frame buffer
Free frame buffer structure
Return IOM_COMPLETED on success else return suitable error code

Figure 18. FVID_FREE_BUFFER flow diagram

3.3.3.5 Channel deletion

The application once it has completed all the transaction can close the channel so that all resources allocated to the channel are freed. To close the channel application calls FVID_delete() (corresponding GIO function GIO_control() API) which in turn calls VPIF IO mini driver's vpifMdDeleteChan() function internally. The Vpif driver provides the vpifMdDeleteChan() API to delete a previously created VPIF channel.

The typical activities performed during the channel creation are as follows:

- The channel to be deleted is in opened state (*Vpif_DriverState_OPENED*) and is not started (*Vpif_DriverState_STARTED*).
- Close external device.
- Delete tasklet handler.
- For display mode set *gVpifRawSelVbiParam* and *gVpifRawSelHbiParam* to default state.
- Delete the queues.
- Disable the channel and disable the channel interrupt.
- Restore channel handle variables to default state.
- The status of the channel is updated to close (*Vpif_DriverState_CLOSED*).

❑ *Once the channel is closed it has no life and is no longer available for further transaction. The user will have to bring the driver back to life by creating the driver through vpifMdCreateChan().*

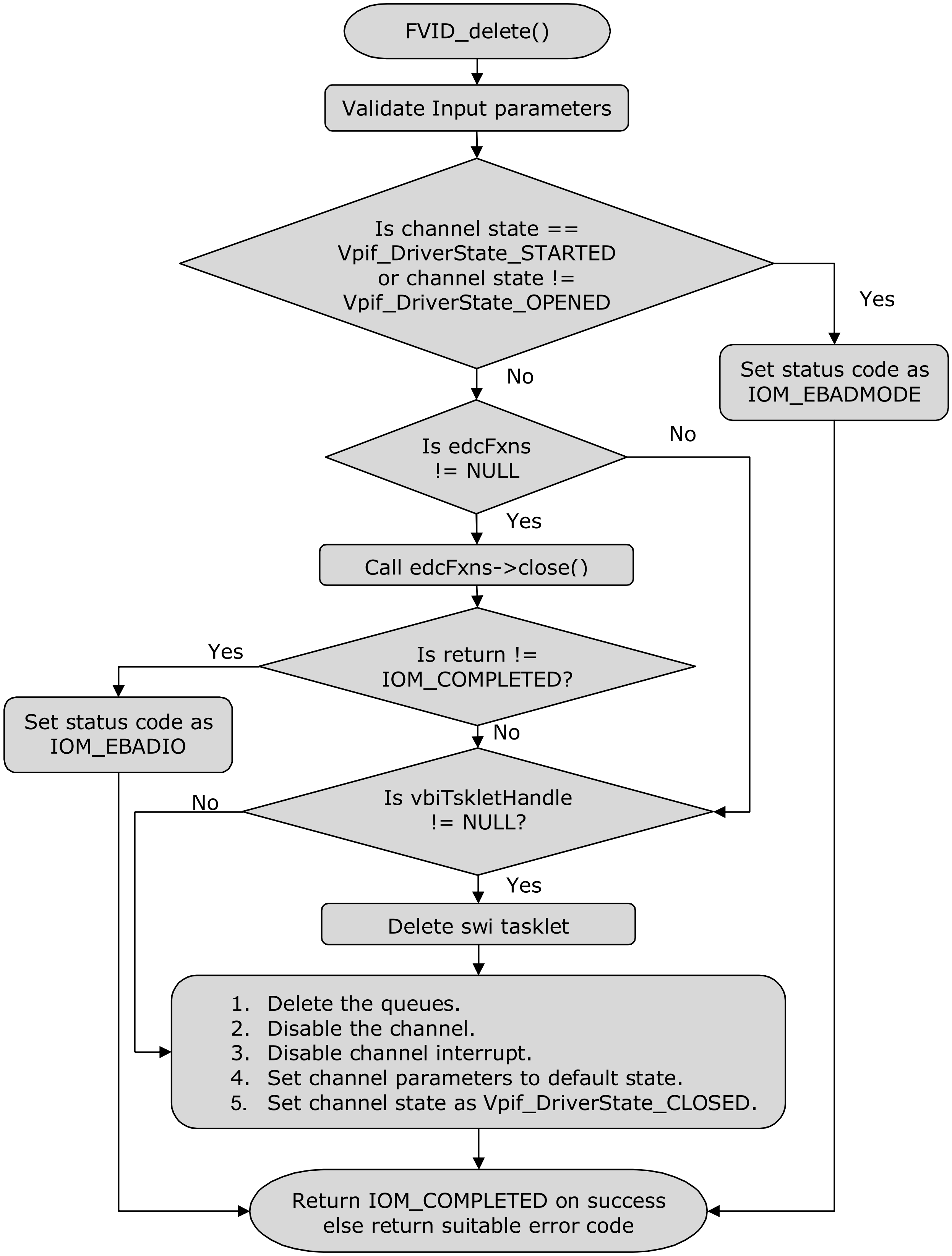

FVID_delete()
Validate Input parameters
Is channel state == Vpif_DriverState_STARTED or channel state != Vpif_DriverState_OPENED
Yes
Set status code as IOM_EBADMODE
No
Is edcFxns != NULL
No
Yes
Call edcFxns->close()
Is return != IOM_COMPLETED?
Yes
Set status code as IOM_EBADIO
No
Is vbiTskletHandle != NULL?
No
Yes
Delete swi tasklet
1. Delete the queues.
2. Disable the channel.
3. Disable channel interrupt.
4. Set channel parameters to default state.
5. Set channel state as Vpif_DriverState_CLOSED.
Return IOM_COMPLETED on success else return suitable error code

Figure 19. Driver channel deletion flow diagram

3.3.3.6 *Driver unbinding/deletion*

VPIF driver provides an interface for deleting the driver. vpifMdUnBindDev() function de-allocates all the resources allocated to the driver during the driver binding operation. This function is called during the closing of the instance. The function does a final clean up before the driver could be relinquished of any use. Here, all the resources which were allocated during instance initialization shall be unallocated; interrupt handlers shall be unregistered, instance specific information set to default values and the state of the instance is set as deleted.

The typical operations performed by the unbind operation are as listed below.

- Check if all the channels are closed.
- Unregister interrupts.
- Set the status of the instance to *Vpif_DriverState_CLOSED*.
- Set the status of the module "*inUse*" to FALSE (so that it can be used again).

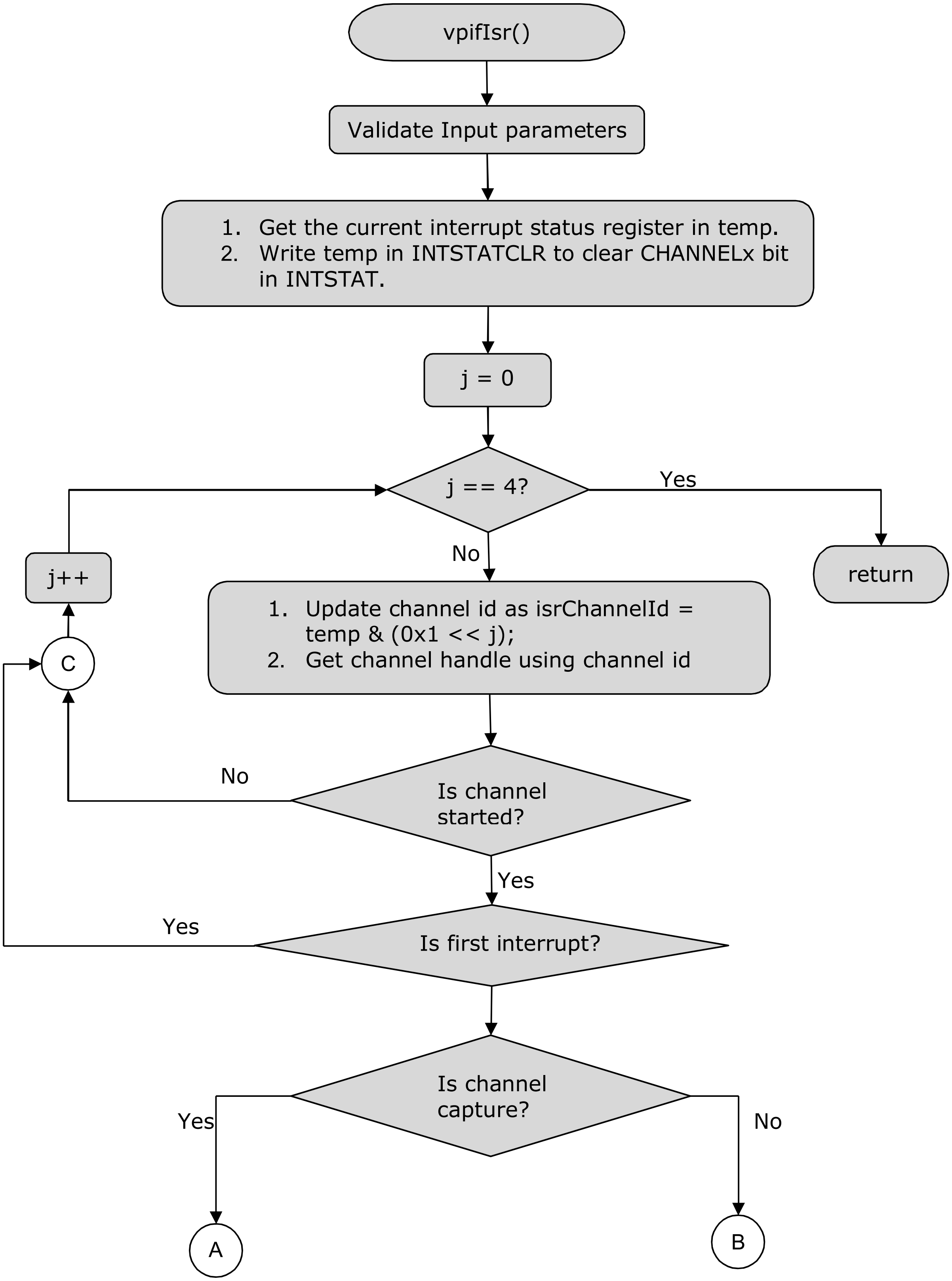
❑ *Following the call vpifMdUnBindDev(), the instance is no more valid and user needs to restart from beginning over an vpifMdBindDev() call to bring driver back to life.*

3.3.3.7 *ISR Handling*

This function is the ISR for the entire Vpif device. vpifIsr() is called when VPIF generates any interrupt. This routine further processes the exact cause of interrupt by reading interrupt status register. It changes status of the displayed or captured buffer, takes next buffer from the queue and sets its address in VPIF registers.

Once the device has been opened and application wants to capture/display the frame it will issue a FVID_queue() call and the buffer gets queued inside the INPUT buffer queue using vpifQueueBuffer(). Once the capture or display operation starts at some point in time, and when the H/W interrupt is asserted, vpifIsr() is called. In the ISR context, vpifIsr(), the driver checks for any new frame buffer in the INPUT queue and if there is any then driver puts the last displayed frame buffer in the OUTPUT queue or if it is a capture operation, it makes the last captured frame as most recent IOP. It deletes entry of frame buffer from the INPUT queue. The ISR also updates the new buffer pointers in the VPIF registers appropriately.

When application wants to do FVID_exchange(), it provide a buffer to be queued inside and this buffer is exchanged with the recently captured buffer or already displayed buffer. If the buffer is available it is exchanged, else it returns *IOM_PENDING*. Once the ISR comes, it checks if there is a request pending. If yes, then the application specific callback functions shall be invoked from the Interrupt context. When a mini-driver completes its processing, usually in an ISR context, it calls its registered callback function to pass the frame buffer back to the user. The callback function is registered with the driver object during channel creation. The FVID_exchange()/callback function pair handles the passing of IO packets between the application and the VPIF IOM layer of the driver.


vpifIsr()
Validate Input parameters
1. Get the current interrupt status register in temp.
2. Write temp in INTSTATCLR to clear CHANNELx bit in INTSTAT.
j = 0
j == 4?
Yes
return
No
j++
1. Update channel id as isrChannelId = temp & (0x1 << j);
2. Get channel handle using channel id
C
No
Is channel started?
Yes
Yes
Is first interrupt?
Is channel capture?
Yes
No
A
B

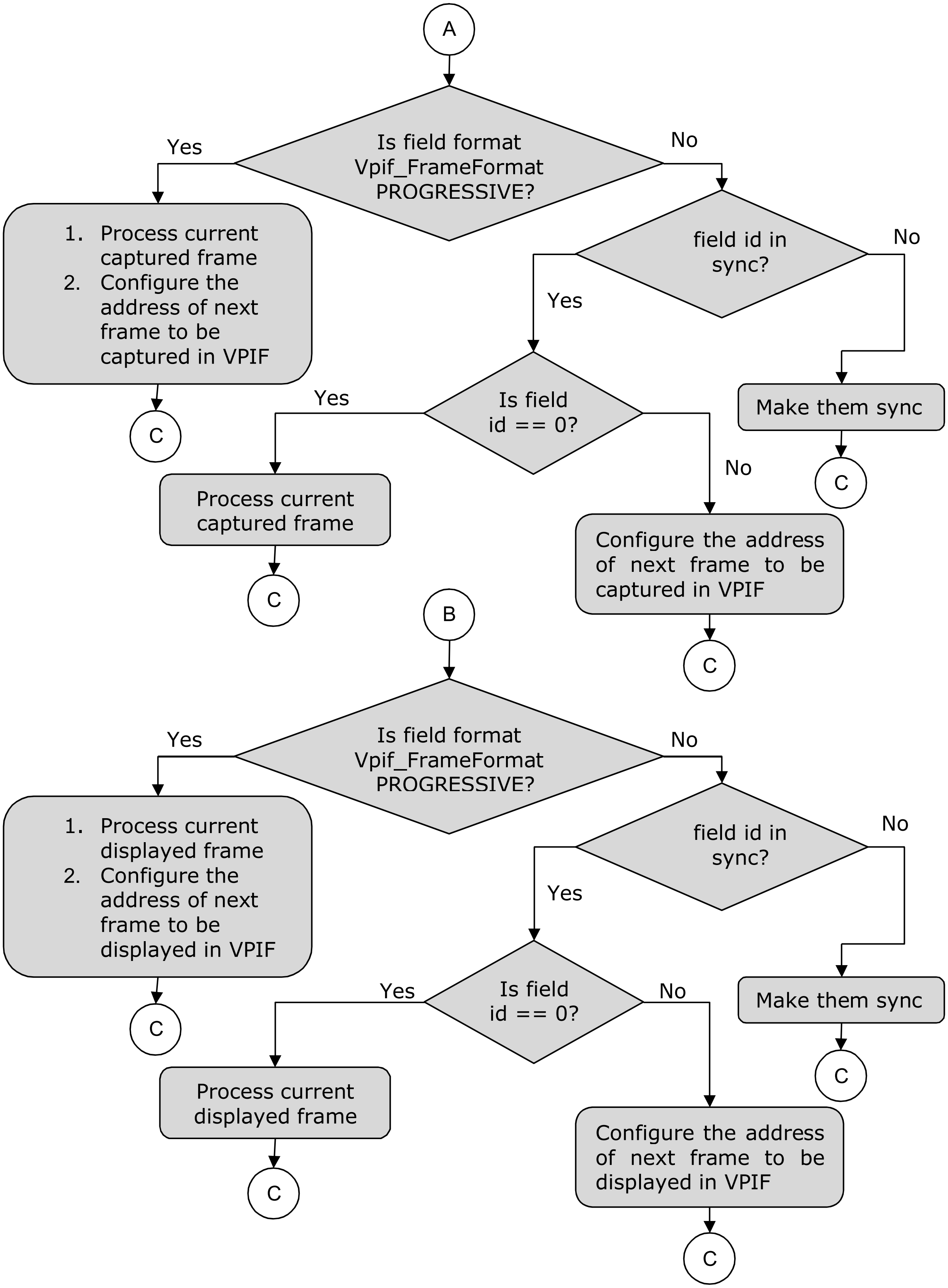

A
Is field format Vpif_FrameFormat PROGRESSIVE?
Yes
No
1. Process current captured frame
2. Configure the address of next frame to be captured in VPIF
C
field id in sync?
Yes
No
Is field id == 0?
Yes
No
Process current captured frame
C
Configure the address of next frame to be captured in VPIF
C
Make them sync
C
B
Is field format Vpif_FrameFormat PROGRESSIVE?
Yes
No
1. Process current displayed frame
2. Configure the address of next frame to be displayed in VPIF
C
field id in sync?
Yes
No
Is field id == 0?
Yes
No
Process current displayed frame
C
Configure the address of next frame to be displayed in VPIF
C
Make them sync
C

Figure 20. ISR flow diagram

3.3.4 Capture Channel Working

The capture driver uses following objects to implement the driver design. It maintains the following variables for each channel object:

- **curViop**: Buffer DMA is moving data into. A current frame that DMA is transferring data into.
- **nextViop**: Buffer DMA is going to move data to after capture completion of current frame. A next frame that DMA is going to work on next.
- **mrViop**: Buffer containing data of the most recently captured frame. A most recent frame for the class driver to grab.
- **qIn**: Queue object that contains empty buffers. A free queue for queuing up empty frame buffers from application.

When a frame has been captured, the previous "most recent frame" is recycled back to the free queue. The newly captured frame then becomes the "most recent frame" or it is passed directly to the class driver in case of a pending I/O request. The class driver always gets the "most recent frame". The class driver blocks when the most recent frame is not available while the I/O request is pending. When the free queue is empty and the current frame is the same as the next frame, the driver holds on to the frame and keeps capturing data into it.

3.3.4.1 Buffer Allocation and Management

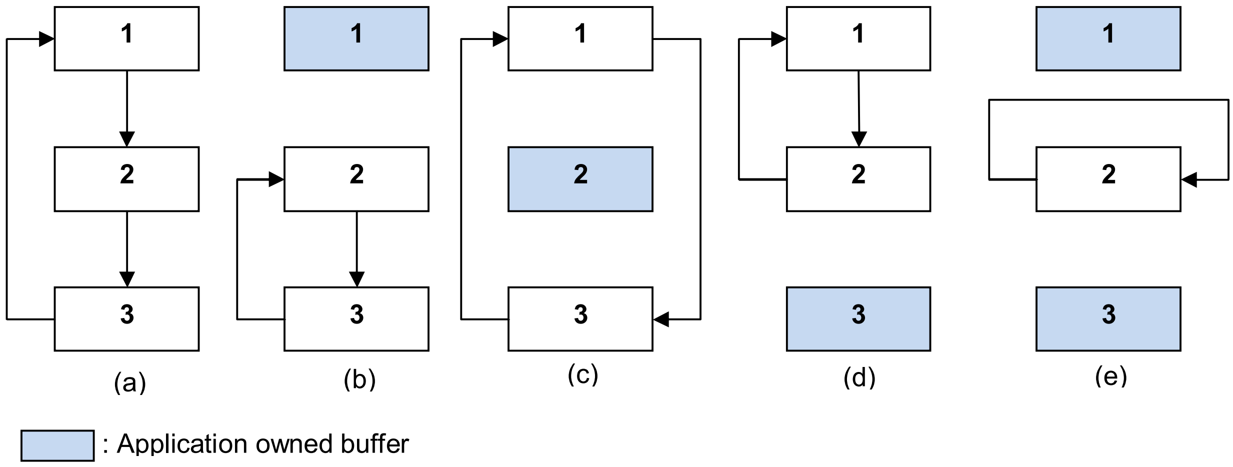


Figure 21. Capture Driver Buffer Management

All buffers are initially in the free queue and the driver cycles through them in a circular fashion. This is illustrated in (a).

When the application calls FVID_dequeue() and grabs the buffer with the most recent data from the driver, the driver then cycles through the rest of buffers. This is illustrated in figure from (a) to (b) and from (b) to (e).

When the application calls FVID_queue(), an empty buffer is returned by the application to the driver's free queue. This is illustrated in figure from (b) to (a) or from (e) to (b).

When the application calls FVID_exchange(), an empty buffer is returned by the application to the driver's free queue, and a buffer with the most recent data is given to the application. This is equivalent to calling FVID_queue() and FVID_ dequeue() sequentially, as shown in figure from (b) to (c) and from (c) to (d).

3.3.5 Display Channel Working

The display driver uses following objects to implement the driver design. It maintains the following variables for each channel object:

- **curViop**: Buffer DMA is moving data from. A current frame that DMA is transferring data out of.
- **nextViop**: Buffer DMA is going to move data from after display completion of current frame. A next frame that DMA is going to work on next.
- **qIn**: Queue object that contains buffers to be displayed. An input queue for queuing up frame buffers that is going to be displayed
- **qOut**: Queue object that contains buffers already displayed. An output queue for queuing up frame buffers that have already been displayed.

The driver maintains two buffer queues, one for incoming buffers, which contains set of buffers ready to be displayed and other is outgoing buffer queue, which are set of buffer already displayed. Driver takes a buffer from the incoming queue and puts it for the display. After buffer display is complete, it will put this buffer in the outgoing queue. *FVID_DEQUEUE* IOCTL will remove the buffer from the driver's outgoing queue. If there are no buffers available in the driver's outgoing queue, it will block until one is available. In the same way, if there are no buffers in the driver's incoming queue, driver will continue to display the last buffer. So driver always keeps one buffer with itself without stopping the display. As it is keeping last buffer, this last buffer cannot be de-queued and call to *FVID_DEQUEUE* IOCTL for the last buffer will be blocked until a buffer is available in driver's incoming queue.

When a frame has been displayed, it goes into the output queue or passed directly to the class driver in case of a pending I/O request.

When the input queue is empty and the current frame is the same as the next frame, the driver holds on to the frame and keeps displaying it.

Once the start request comes the driver starts to display the data pointed by the *curViop*. As soon as the channel operation starts an interrupt is asserted, which marks the start of DMA operation for that channel. Once the frame or top field interrupt comes, update the *curViop* with buffer pointing to *nextViop*. Also update the *nextViop* by de-queuing the buffer from the *qIn*, if queue is not empty, otherwise do not update the *nextViop* (i.e. hold on earlier buffer).

If the currently displayed packet is same as the *curViop* then do not enq it in the *qOut* and keep on displaying it. If they are not same then check if there is an outstanding request pending or not. If yes the directly update the packet and call the callback function else enqueue it in the *qOut*.

3.3.5.1 *Buffer Allocation and Management*

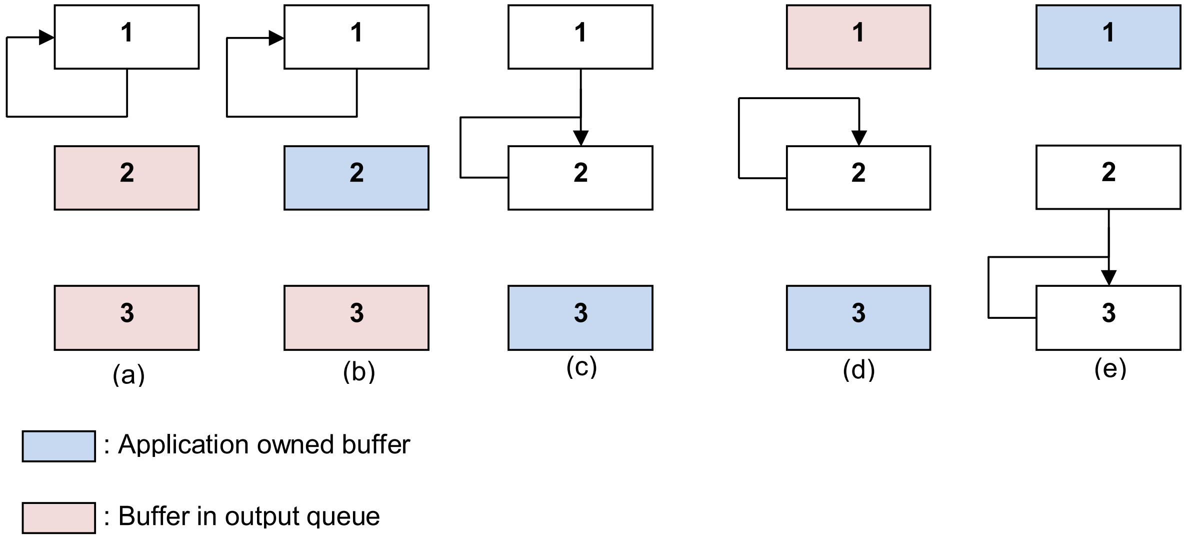


Figure 22. Display Driver Buffer Management

Initially all buffers except one are in the output queue, ready to be grabbed by the application. The driver repeatedly displays the current buffer. This is shown in figure (a).

When the application calls FVID_dequeue(), it gets a buffer from the driver. Application starts to fill data to it while the driver is still displaying its current buffer. This is shown in figure (a) to (b).

When the application calls FVID_queue(), it returns a buffer ready for display back to the driver. The driver, in turn, will set this buffer as its current buffer after it completes displaying the previous one. This is shown in figure (b) to (c) to (d).

When the application calls FVID_exchange(), it returns a buffer ready for display back to the driver and it requires an empty buffer from the driver. This is equivalent to calling FVID_queue() and FVID_dequeue() sequentially, as shown in figure (d) to (e).

3.3.6 Slice VBI Handling

Slice VBI service is enable by the application during channel creation. Once the service is enabled, the buffers are allocated for the services during normal buffer allocation mechanism. Once the buffers are allocated the frame buffer contains the slice VBI data side by side with the video data. For handling the slice VBI data two tasks for capture and display are created. The call for read or write of slice VBI data takes place for each filed of the frame. Following sections tells about how the slice VBI data is handled inside the driver.

3.3.6.1 *Display Slice VBI Handling*

For Display, during EDC open of the EDC device the VPIF driver will register a callback function and a callback handle. This will be used whenever the I2C write to the I2C driver is complete.

Slice VBI display, once enable, will call to the ADV7343 driver for writing the slice VBI data. This call will, be made whenever the field ISR comes to the driver. From the ISR the driver posts *syncSem* to the task. Once the task receives the semaphore the driver calls the ADV7343 to write the Slice VBI data. The ADV7343 driver calls the codec interface layer telling the callback function of ADV7343 to be called once the call completed. The codec interface layer has only two packets that can be submitted to the I2C driver. It takes a packet form *txFreeQ*, updates the parameters for packet and calls the I2C driver. This is a non blocking call to the I2C driver and will return immediately once the packet is submitted to the driver.

Whenever this packet is transferred, the I2C driver will call the callback function of the codec interface layer. The codec interface layer puts the packet back to the *txFreeQ* and calls the callback function of the ADV7343 layer. The ADV7343 layer has the callback function of VPIF, given during EDC open, and calls it. In this callback function the VPIF driver updates the status of the write callback and post a SWI. Once the SWI is posted, if it is field 0 process the current display frame(If a request is pending release the frame else put it in the out queue) and if this is a field 1 configure the next display frame pointer in the VPIF(update the VPIF registers with the new frame pointers)

Following sequence diagram will elaborate the same:

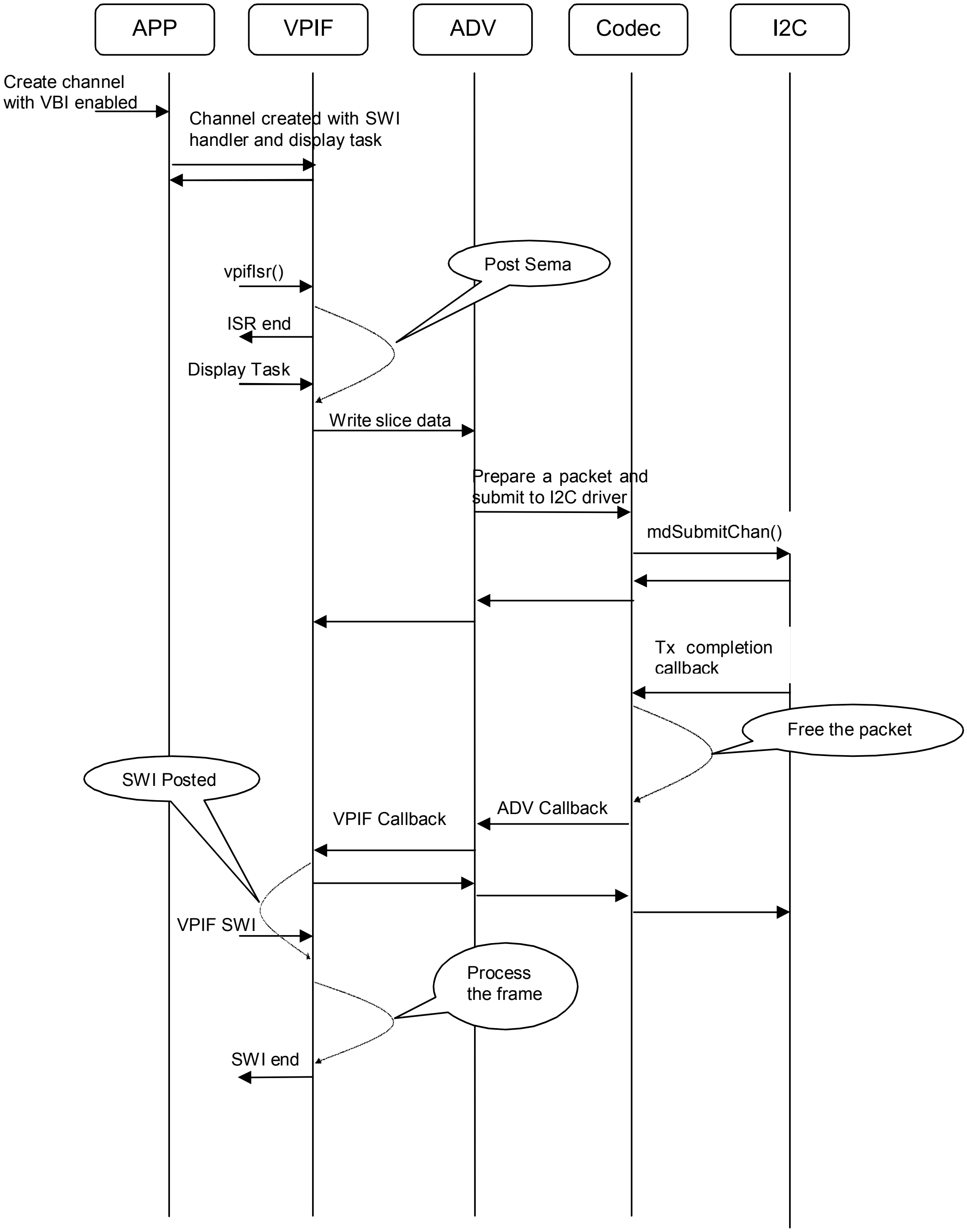


Figure 23. Display Slice VBI Handling

3.3.6.2 *Capture Slice VBI Handling*

For capture slice data, the read calls are completed within the calling context. A task is created for capturing the slice VBI data. This task is created and it pends on `syncSema`.

Whenever a field ISR comes for capture, it first posts the *syncSema* on which the task is pending for reading the slice VBI data. As soon as the semaphore is posted the task reads the VBI data from the TVP5147. The TVP5147 driver calls the codec interface layer, which in turn submits the I2C packet to the I2C driver. This is a blocking call to the I2C driver and the call waits on a semaphore. This semaphore is released once I2C callback for the transfer comes.

Once the read of slice data completes, if it is field 0 process the current captured frame (If a request is pending release the frame else make it as most recent IO packet) and if this is a field 1 configure the next display frame pointer in the VPIF (update the VPIF registers with the new frame pointers).

Following sequence diagram will elaborate the same:

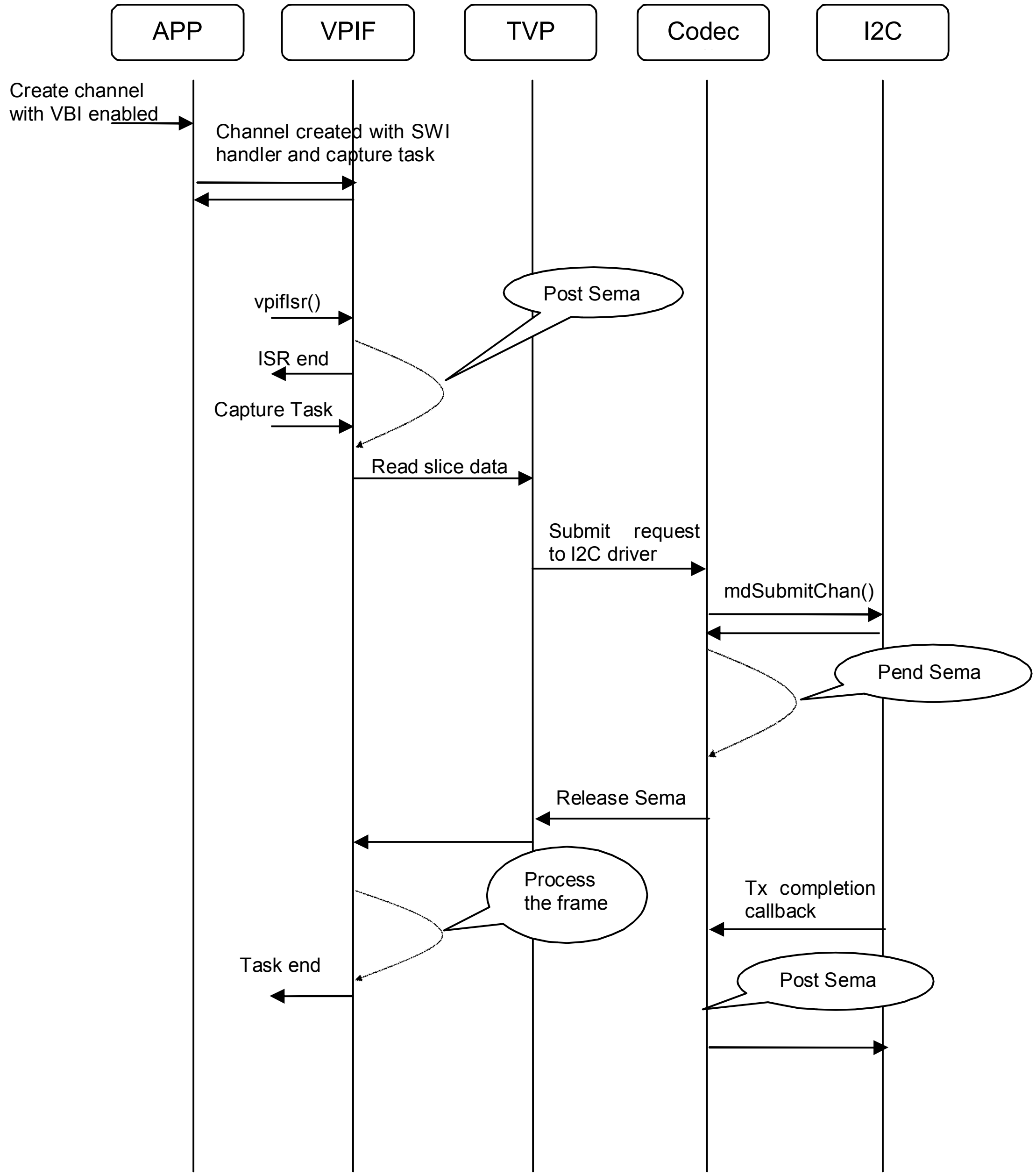


Figure 24. Capture Slice VBI Handling

3.3.7 Data Flow

3.3.7.1 Hardware Data Flow

The following diagram shows the data and control flow in hardware, during capture and display operations.

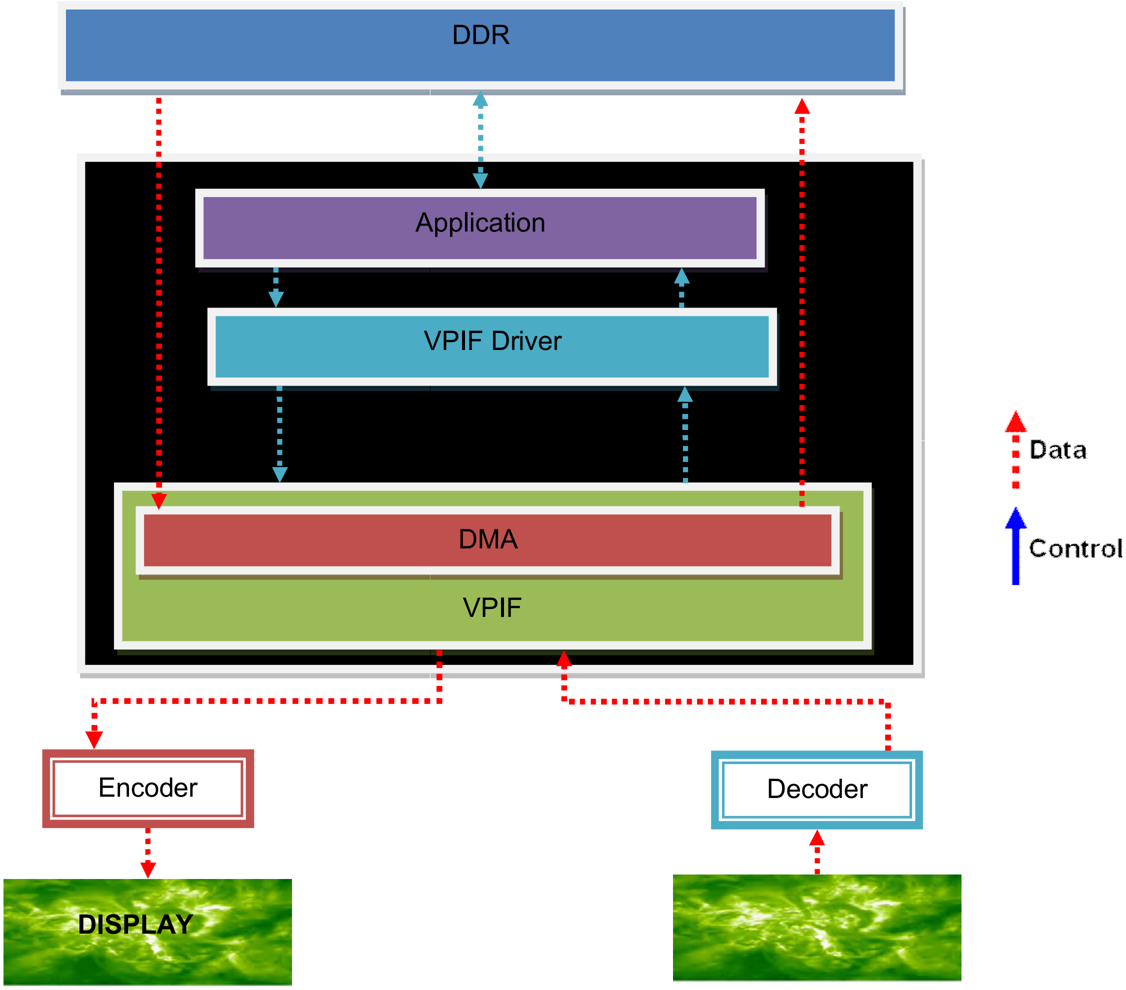


Figure 25. Hardware data flow

3.3.7.2 Software Data Flow

The following diagram shows the data flow in software, during capture and display operations.

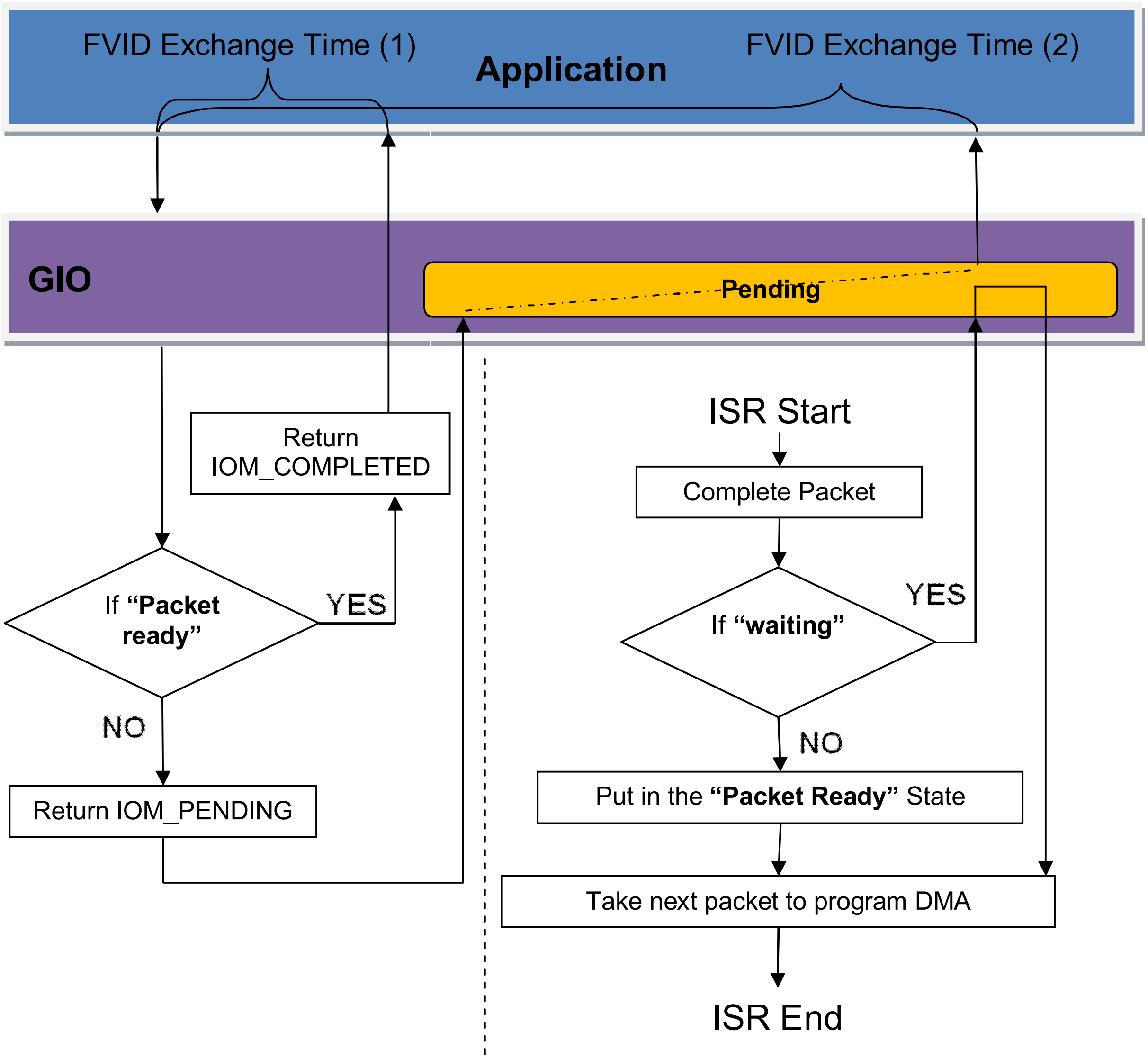


Figure 26. Software data flow

3.4 EDC driver

This section describes in detail about External Device Control (EDC) mechanism of VPIF driver - EVM or hardware dependent components that are not built inside VPIF module and VPIF has dependency on such peripherals. OMAPL138 video port driver configures external video decoders and encoders using I2C interface to capture or display video. This section describes the functions, data structures and enumerations for the EDC module. Most of the functionality and features supported by the EDC driver depends on the OMAPL138EVM schematics and VPIF support. Features which are not supported by the current OMAPL138EVM and VPIF are mentioned as NOT SUPPORTED in the appropriate places. The options which are not supported are given only for future purpose.

3.4.1 Interface between VPIF and EDC Driver

Below figure shows interface between VPIF driver and EDC driver when any function is being called from application. Here, EDC Open, EDC Control or EDC Close functions represent corresponding encoder/decoder functions.

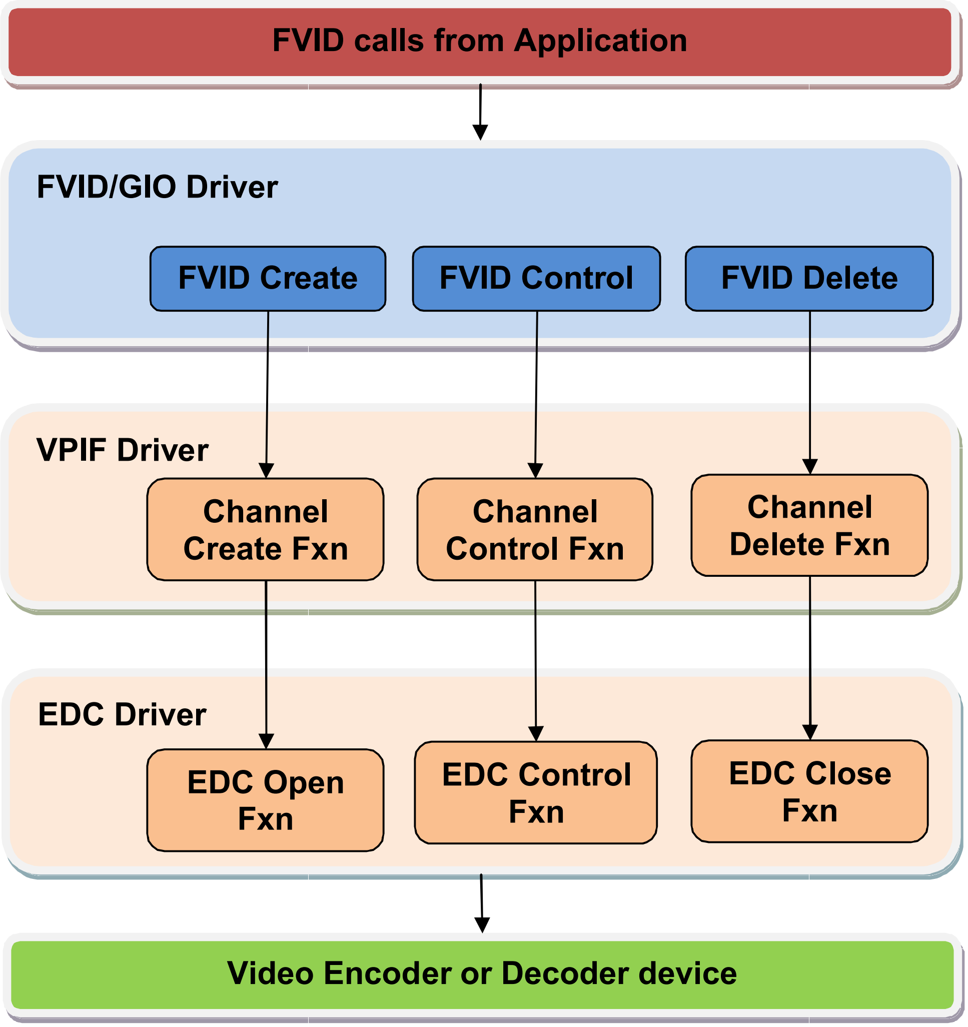


Figure 27. Interaction between VPIF and EDC driver

The EDC driver is associated with each channel of the VPIF driver through the *edcTbl* member (of type *EDC_Fxns*) of *Vpif_CapChanParams* or *Vpif_DisChanParams*. This is passed during VPIF driver channel creation call to vpifMdCreateChan(). Each VPIF channel can be associated with one EDC drivers.

⚐ *If edcTbl is NULL then it is assumed that the channel has no external encoder or decoder attached.*

Below structure definition provides details about the function pointers where-in the external encoder/decoder plugs-in.

```
typedef struct EDC_Fxns_t

{
    EDC_Handle (*open)(String name, Ptr optArg);

    /**< edcOpen() - required, open the device */

    Int32 (*close)(Ptr devHandle);

    /**< edcClose() - required, close the device */

    Int32 (*ctrl)(Ptr devHandle, Uns cmd, Ptr arg);

    /**< edcCtrl() - required, control/query device */

}EDC_Fxns;
```

⚐ *Every EDC based encoder /decoder/sensor should export its function table pointer through xxx_Fxns global variable.*

The OMPAL138 EVM has the following external encoders and decoders. The details of each driver interface are explained in the user guide.

- Two TVP5147 Decoders
- One ADV7343 Encoder
- External MT9T001 Sensor

3.4.1.1 EDC Open

When application calls FVID_create(), VPIF driver will internally calls the respective xxx_open function. This will power on the EDC device and initialize I2C driver for serial communication. The EDC device name, I2C device to be used and the EDC device address should be passed as argument to xxx_open function to open the corresponding EDC.

☐ *Generally a default configuration is provided by the EDC drivers in which the device is opened. If user wants to use some different configuration, it can be changed using `Edc_IOCTL_CONFIG` ioctl. To find the default configuration of EDC driver, please refer user guide.*

3.4.1.2 EDC Control

To configure EDC, application has to call FVID_control() function with `Vpif_IOCTL_CMD_MAX` + EDC IOCTLs as command. Vpif driver will internally call xxx_ctrl function.

⚐ *EDC ioctls could be invoked using FVID_control() calls on the Vpif driver. These ioctl goes to the respective EDC associated with the channel. Some ioctls are standard and need to be implemented by all EDC devices (with some restriction, if any). These IOCTL are defined in `Edc.h`. The format of IOCTL command to be passed is (`Vpif_IOCTL_CMD_MAX` + command number of the underlying EDC layer). However, some IOCTL commands may be specific for that particular EDC and these ioctls are defined in their respective files. In such cases the ioctl command to be passed is (`Vpif_IOCTL_CMD_MAX` + `Edc_IOCTL_CMD_MAX` + specific command number of the underlying EDC layer). For example, `Edc_IOCTL_CONFIG` is a standard command and will set the configuration of EDC device. However, enabling the color bar in ADV7343 is specific to the ADV7343 encoder. This IOCTL is defined in `Adv7343.h`. The command number for this should be passed as (`Vpif_IOCTL_CMD_MAX` +*

`Edc_IOCTL_CMD_MAX` + `Adv7343_IOCTL_ENABLE_COLORBAR`). These IOCTLs are routed appropriately to the underlying EDC layer.

- ❑ *For different generic/specific EDC IOCTL command please refer to the user guide.*

3.4.1.3 EDC Close

Once the VPIF driver deletes the channel, it will delete the EDC driver instance associated with that channel and close the I2C driver as well. FVID_delete() internally calls xxx_close function of the EDC driver associated with the channel.

3.4.2 Codec Interface

The codec interface provides an abstraction layer between the codec (encoder or decoder) driver and the underlying hardware codec. The codec interface helps in having a uniform view of the underlying codec. This helps as codec driver do not worry about the control bus configuration and configuring of the codec.

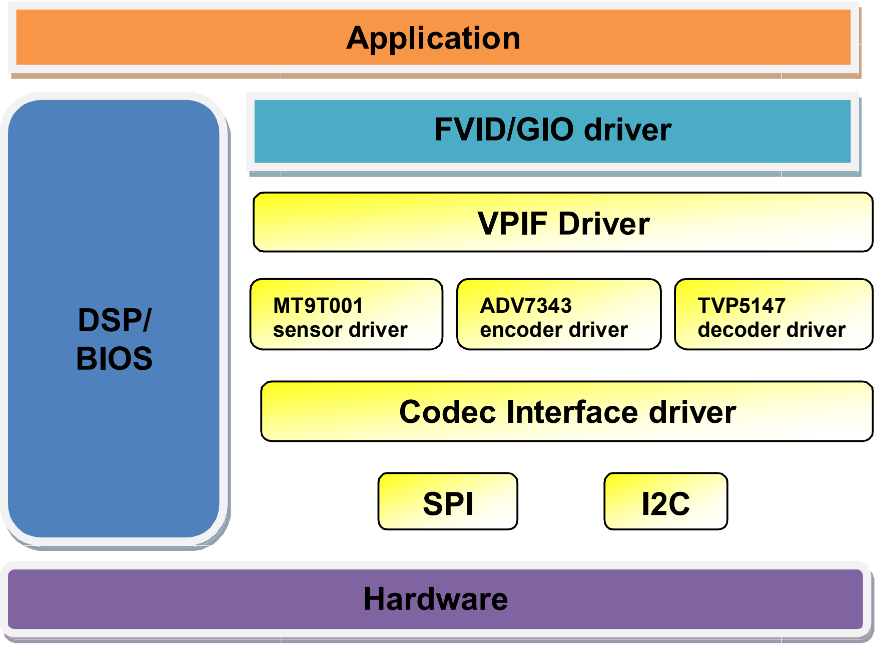


Figure 28. Interaction between EDC drivers and codec interface

The above figure shows the functional composition of the codec interface driver. The codec driver can be configured using a control bus. Two types of control buses are possible:

- SPI bus.
- I2C bus.

The usage of different control buses is dependent on the hardware support available and layout.

The codec interface only supports I2c control bus. Trying to configure through the SPI bus will return "`IOM_ENOTIMPL`" i.e. not implemented as error.

Following are the interfaces implemented:

- initCodecInterface()

  This function is called by the EDC driver during its open call. This in turn opens the codec interface. The control bus required to be open is passed as a string parameter from application to Vpif driver, which is again passed to the underlying EDC driver during open. The EDC driver during open parses the string and pass the required control device string parameter as "*devName*" to the initCodecInterface() function. The codec interface matches the "*devName*" using DEV_match() and finally open the control bus to the codec. Now the codec input and output handle are updated.

- codecWriteReg()

  This function writes to the codec registers using the control bus.

- codecReadReg()

  This function reads the codec registers using the control bus.

- deInitCodecInterface()

  This function de-initializes the underlying codec bus and frees all the resources previously allocated.

4 Low Level Definitions

The mini-driver employs the *Vpif_Object* and *Vpif_ChanObj* structures to maintain internal parameters, state etc. of the instance and channel respectively.

In addition, the driver has two other structures defined – the device params and channel params. The device params structure is used to pass parameters to initialize the driver during DSP/BIOS™ initialization. The channel params structure is used to specify required characteristics while creating a channel.

The interaction between the application and the device is through the instance object and the channel object. While the instance object represents the actual hardware instance, the channel object represents the connection of the application with the driver (and hence the device) for that particular IO direction. It is the channel which represents the characteristic/types of connection the application establishes with the driver/device and hence determines the data transfer capabilities the user gets to do to/from the device. For example, the channel could be input/output channel. This capability provided to the user/application, per channel, is determined by the capabilities of the underlying device.

The following sections provide data structures maintained by IOM interface internally.

□ *For details and usage of other data structures, exposed to user, can be found in the User guide.*

4.1 Constants & Enumerations

4.1.1 VPIF_SD_PARAMS

This macro defines the various SD video parameters supported. It contains value of SD parameters of type *Vpif_ConfigParams*. These parameters are required to program the VPIF internal registers.

Definition

```
#define VPIF_SD_PARAMS \
    {Vpif_VideoMode_NTSC, 720, 480, 30, Vpif_FrameFormat_INTERLACED, \
     Vpif_YCMuxed_YES, 268, 1440, 1, 23, 263, 266, \
     286, 525, 525, Vpif_CaptureFormat_BT, TRUE, FALSE, \
     720, VPIF_NTSC_HBI_PARAMS, VPIF_NTSC_VBI_PARAMS}, \
    {Vpif_VideoMode_PAL, 720, 576, 25, Vpif_FrameFormat_INTERLACED, \
     Vpif_YCMuxed_YES, 280, 1440, 1, 23, 311, 313, \
     336, 624, 625, Vpif_CaptureFormat_BT, TRUE, FALSE, \
     720, VPIF_PAL_HBI_PARAMS, VPIF_PAL_VBI_PARAMS}
```

Comments

Values defined by this macro set the look up table required inside by the driver. This table is used if the user does not supply the *videoParams* parameter during channel creation.

Constraints

None

See Also

`Vpif_ConfigParams, Vpif_DisChanParams, Vpif_CapChanParams`

4.1.2 VPIF_MT9T001_PARAMS

This macro defines the parameters for various resolutions supported by the MT9T001 device. The values are of type *Vpif_ConfigParams*. These parameters are required to program the VPIF internal registers.

Definition

```
#define VPIF_MT9T001_PARAMS \
    {Vpif_VideoMode_RAW_VGA, 640, 480, 93, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
    {Vpif_VideoMode_RAW_SVGA, 800, 600, 65, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
    {Vpif_VideoMode_RAW_XGA, 1024, 768, 43, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
    {Vpif_VideoMode_RAW_SXGA, 1280, 1024, 27, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
    {Vpif_VideoMode_RAW_UXGA, 1600, 1200, 20, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
    {Vpif_VideoMode_RAW_QXGA, 2048, 1536, 12, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
    {Vpif_VideoMode_RAW_480P, 720, 480, 60, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
    {Vpif_VideoMode_RAW_576P, 720, 576, 50, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
    {Vpif_VideoMode_RAW_720P, 1280, 720, 39, 
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}, \
     {Vpif_VideoMode_RAW_1080P, 1920, 1080, 18,
Vpif_FrameFormat_PROGRESSIVE, \
     Vpif_YCMuxed_NO, 0, 0, 0, 0, 0, 0, \
     0, 0, 0, Vpif_CaptureFormat_CCDC, FALSE, FALSE, \
     0, {0, 0, 0}, {0, 0, 0}}
```

Comments

All the line parameters and vsize are set to 0 as they are not required by VPIF and their value are calculated using the HSYNC and VSYNC.

Constraints

None

See Also

`Vpif_ConfigParams, Vpif_DisChanParams, Vpif_CapChanParams`

4.1.3 VPIF_CHn_MAX_MODES

This macro defines the number of modes supported by different the VPIF channels. N stands for 0, 1, 2 and 3.

Definition

```
#define VPIF_CH0_MAX_MODES                 (12u)
#define VPIF_CH1_MAX_MODES                 (2u)
#define VPIF_CH2_MAX_MODES                 (2u)
#define VPIF_CH3_MAX_MODES                 (2u)
```

Comments

None

Constraints

None

See Also

`None`

4.1.4 Vpif_IoMode

This structure defines different internal IO mode in which a channel can be opened.

Definition

```
typedef enum Vpif_IoMode_t
{
    Vpif_IoMode_NONE,
    Vpif_IoMode_RAW_CAP,
    Vpif_IoMode_CAP,
    Vpif_IoMode_DIS
}Vpif_IoMode;
```

Comments

IOM_INPUT and *IOM_OUTPUT* decide, whether the channel is capture or display. For RAW capture the mode is decided updated using this enum.

Constraints

None

See Also

None

4.2 Typedefs & Data Structures

4.2.1 Driver Instance Object

This structure is the Vpif driver's internal data structure for maintaining VPIF object per-instance state. This data structure is used by the driver to hold the information specific to the instance. There will be one unique instance object for every instance of the VPIF controller supported by the driver.

Definition

```
typedef struct Vpif_Object_t
{
    Int32                      instNum;
    Vpif_DmaReqSize            dmaSize;
    Uint32                     devState;
    Vpif_HwInfo                deviceInfo;
    Bool                       isPwrAware;
    Vpif_PwrmInfo              pwrmInfo;
    Vpif_ChanObj               chanObj[Vpif_TOTAL_NUM_CHANNELS];
}Vpif_Object;
```

Fields

`instNum` Instance number or device ID of the VPIF.

`dmaSize` VPIF DMA Size value.

`devState` Instance state, contains information on whether instance is opened, closed, etc

`deviceInfo` Instance specific hardware information. Structure holding the hardware information related to the instance (e.g. interrupt numbers, base address etc)

`isPwrAware` Is the driver power management aware or not

`pwrmInfo` PWRM related information storage

`chanObj[]` Holds four channel objects for channel 0, 1, 2 and 3

Comments

One instance object represents one instance of driver.

Constraints

None

See Also

Vpif_ChanObj, Vpif_HwInfo, Vpif_DmaReqSize, Vpif_PwrmInfo

4.2.2 Channel object

This structure is the Vpif driver's internal data structure. This data structure is used by the driver to hold the information specific to the channel. There will be at most four channels supported per instance (two for capture and two for display). It is used to maintain the information pertaining to the channel like the current channel state, callback function etc. This structure is initialized by vpifMdCreateChan() and a pointer to this is passed down to all other channel related functions. Lifetime of the data structure is from its creation by vpifMdCreateChan() till it is invalidated by vpifMdDeleteChan().

Definition

```
typedef struct Vpif_ChanObj_t
{
    Uint32                                  chState;
    Ptr                                     devHandle;
    Uint32                                  chError;
    Uint32                                  frameInt;
    Uint8                                   chanNum;
    CSL_VpifRegsOvly                        baseAddress;
    Vpif_SdramStorage                       frameStorageMode;
    IOM_TiomCallback                        cbFxn;
    Ptr                                     cbArg;
    Vpif_VideoMode                          vidMode;
    Vpif_IoMode                             channelMode;
    EDC_Fxns                                *edcFxns;
    Ptr                                     edcHandle;
    Int32                                   segId;
    Uint32                                  bufSz;
    Uint16                                  hpitch;
    Uint16                                  lineOffset;
    Uint16                                  alignment;
    Uint8                                   started;
    Bool                                    useTwoChannels;
    Uint32                                  fieldId;
    Vpif_ConfigParams                       chVidParams;
    QUE_Obj                                 qIn;
    QUE_Obj                                 qOut;
    volatile Bool                           queEmpty;
    FVID_Frame                              *curViop;
    FVID_Frame                              *nextViop;
    FVID_Frame                              *mrViop;
    IOM_Packet                              *packetIOM;
    Int32                                   numQueBuf;
    Uint32                                  hbiBufSz;
    Uint32                                  vbiBufSz;
    Ptr                                     vbiTskletHandle;
    Int32                                   sliceServType;
    Uint8                                   numSliceServices;
    Int32                                  sliceVbiFld0I2cSubmitStatus;
    Int32                                   sliceVbiFld0I2cCbStatus;
    Int32                                  sliceVbiFld1I2cSubmitStatus;
    Int32                                   sliceVbiFld1I2cCbStatus;
    Int32                                   currPktStatus;
    TSK_Handle                              taskHandle;
    Bool                                    taskRun;
    SEM_Handle                              syncSem;
    Vpif_RawSelectiveVbiParams              *setSelVbiDispParams;
    Vpif_RawSelectiveVbiParams              *setSelHbiDispParams;
    Vpif_RawCaptureDataWidth                dataSz;
    Vpif_RawCapturePinPol                   fieldPolarity;
    Vpif_RawCapturePinPol                   vPixelPol;
    Vpif_RawCapturePinPol                   hPixelPol;
}Vpif_ChanObj;
```

Fields

`chState`	Channel status opened, configured, allocated, etc.
`devHandle`	Handle to the Device.
`chError`	Number of Channel errors
`frameInt`	Number of Frame interrupts.
`chanNum`	Channel number of instance.
`baseAddress`	Pointer to VPIF register structure defined in *cslr_vpif.h*. This will point to base address of VPIF.
`frameStorageMode`	Indicates whether it is field or frame based storage mode.
`cbFxn`	IOM callback function.
`cbArg`	IOM callback function argument.
`vidMode`	Video mode supported by the channel.
`channelMode`	Channel’s IO mode of operation.
`*edcFxns`	Pointer to function table of external device.
`edcHandle`	Handle of external device.
`segId`	Memory segment id to heap from where allocation has to be done on application's behalf to allocate frame buffer memory.
`bufSz`	Video Buffer size in bytes.
`hpitch`	Horizontal pitch size in bytes.
`lineOffset`	Line offset for DMA in bytes.
`alignment`	Frame buffer alignment.
`started`	Indicates whether streaming started.
`useTwoChannels`	Indicates whether to use one channel or two.
`fieldId`	Indicates id of the field of the current frame.
`chVidParams`	Video parameters for the channel.

qIn

IN queue associated with this channel. For display this queue object contains buffers to be displayed. For capture this queue object contains empty buffers.

qOut

OUT queue associated with this channel. For display this queue object that contains buffers already displayed. For capture this is not used.

queEmpty

Flag to maintain queue status.

*curViop

Current video IO packet. For display it is the current frame that DMA is transferring data out of. For capture it is the current frame that DMA is transferring data into.

*nextViop

Next video IO packet. For display it is the buffer DMA is going to move data from after display completion of current frame. For capture it is the buffer DMA is going to move data to after capture completion of current frame.

*mrViop

Most recent video IO packet. For capture it is the buffer containing data of the most recently captured frame. For display this is not used.

*packetIOM

Pointer to IOM packet. IOM Packet passed by the GIO layer is stored here.

numQueBuf

Number of submitted buffer.

hbiBufSz

HBI Buffer size in bytes. A value of 0 suggests that RAW HBI is not enabled on the channel.

vbiBufSz

VBI Buffer size in bytes. A value of 0 suggests that RAW HBI is not enabled on the channel.

vbiTskletHandle

Tasklet (Software interrupt) handle.

sliceServType

Type of Slice VBI service. Available values for this field are defined in "*Vpif.h*" file with VPIF Slice VBI service type title. This should be passed appropriately according to the Video standard mode desired.

numSliceServices

Number of slice VBI services enabled. A value of 0 suggests that Slice VBI is not enabled for the channel.

sliceVbiFld0I2cSubmitStatus

Slice VBI field 0 submit status for I2C

sliceVbiFld0I2cCbStatus

Slice VBI field 0 callback status for I2C

sliceVbiFld1I2cSubmitStatus

Slice VBI field 1 submit status for I2C

sliceVbiFld1I2cCbStatus

Slice VBI field 1 callback status for I2C

currPktStatus	Packet status of the current channel
taskHandle	VPIF Slice VBI Task Handle
taskRun	VPIF Slice VBI Task running or not
syncSem	Semaphore for synchronization between task and ISR
*setSelVbiDispParams	Place holder for Selective positioning of RAW VBI Display data. If set to NULL implies no selective positioning required.
*setSelHbiDispParams	Place holder for Selective positioning of RAW HBI Display data. If set to NULL implies no selective positioning required.
dataSz	The data width bit is only used with the CCD/CMOS data capture mode.
fieldPolarity	Field ID polarity inverting control.
vPixelPol	Vertical pixel valid signal polarity control.
hPixelPol	Horizontal pixel valid signal polarity control.

Comments

- Only four channels are supported per instance.
- Raw capture should only use channel 0 and channel 1 cannot be used.

Constraints

None

See Also

Vpif_SdramStorage, Vpif_VideoMode, Vpif_IoMode, EDC_Fxns, Vpif_ConfigParams, FVID_Frame, Vpif_RawSelectiveVbiParams, Vpif_RawCaptureDataWidth, Vpif_RawCapturePinPol

4.2.3 Vpif_Params

This structure is used to supply user parameters during the creation of the driver instance. During the creation of the driver user needs to supply the above structure with the required parameters.

Definition

```
typedef struct Vpif_Params_t
{
    Uint16                              hwiNumber;
    Vpif_DmaReqSize                     dmaReqSize;
    Bool                                pscPwrmEnable;
}Vpif_Params;
```

Fields

hwiNumber	ECM event group of VPIF instance.
dmaReqSize	Video FIFO threshold value.

pscPwrmEnable	Control for module power management enable & disable. Set it to TRUE, if user wants the driver to take care of power management. Set it to FALSE, if power is always enabled by application.

Comments

None

Constraints

None

See Also

None

4.2.4 Vpif_DisChanParams

Structure used to supply the display channel parameters during the creation of display channel. During the creation of the channel, user needs to supply the above structure with the appropriate parameters as per the mode of operation.

Definition

```
typedef struct Vpif_DisChanParams_t
{
    Vpif_VideoMode                      dispStdMode;
    Vpif_IoMode                         dispChannelIoMode;
    Vpif_FrameBufferParams              dispFbParams;
    Vpif_SdramStorage                   dispStorageMode;
    EDC_Fxns                            *dispEdcTbl;
    Vpif_ConfigParams                   *dispVideoParams;
    Int32                               dispVbiService;
    Int32                               dispVbiSliceService;
    Vpif_RawSelectiveVbiParams          *dispVVbiParams;
    Vpif_RawSelectiveVbiParams          *dispHVbiParams;
}Vpif_DisChanParams;
```

Fields

dispStdMode	Operation mode title.
dispChannelIoMode	Operation mode for which the channel is opened
dispFbParams	Frame buffer settings.
dispStorageMode	Indicates whether it is field or frame based storage mode. This is only applicable for interlaced mode of operation.
*dispEdcTbl	Function table of encoder module for the channel. A statically defined EDC function table is passed to the vpifMdCreateChan() function via the channel parameters argument.

`*dispVideoParams` Specify the Video parameters if application would like to specify them. This is an optional parameter. If not used, set this element to NULL. If set to NULL, the driver will read the parameters depending upon the mode set. If it is not NULL these values will prevail over whatever mode being set. CAUTION: If wrong parameters are sent, the driver does not verify the validity of these parameters

`dispVbiService` Indicates what type VBI services are required by this mode. Available values for this field are defined in "*Vpif.h*" file with VPIF VBI Ancillary Data service title.

`dispVbiSliceService` If the VBI type is Slice VBI then what kind of service it is. Valid only if one of the *vbiService* is set as *Vpif_VbiServiceType_SLICE_VBI*. Whatever slice service is set here only that data is displayed. Available values for this field are defined in "*Vpif.h*" file with VPIF Slice VBI service type title.

`*dispVVbiParams` Indicates the parameters for selective Vertical blanking data. Value of NULL suggests that selective sub-regions in the VBI space are not required. For selectively sub-regions in the VBI space this should hold appropriate value.

`*dispHVbiParams` Indicates the parameters for selective Horizontal blanking data. Value of NULL suggests that selective sub-regions in the VBI space are not required. For selectively sub-regions in the VBI space this should hold appropriate value.

Comments

"*vVbiParams*" and "*hVbiParams*" are available only if selective pacing of raw Vbi data during disaply.

Constraints

None

See Also

`Vpif_RawCapturePinPol, Vpif_IoMode, Vpif_FrameBufferParams, Vpif_VideoMode, Vpif_SdramStorage`

4.2.5 Vpif_CapChanParams

The application passes a data structure *Vpif_CapChanParams* at time of channel creation for capture. During the creation of the channel, user needs to supply the above structure with the appropriate parameters as per his mode of operation.

Definition

```
typedef struct Vpif_CapChanParams_t
{
    Vpif_VideoMode                      capStdMode;
    Vpif_IoMode                         capChannelIoMode;
    Vpif_FrameBufferParams              capFbParams;
    Vpif_SdramStorage                   capStorageMode;
    EDC_Fxns                                   *capEdcTbl;
    Vpif_ConfigParams                          *capVideoParams;
    Int32                                      capVbiService;
    Int32                                      capVbiSliceService;
    Vpif_RawCaptureDataWidth                   capDataSize;
    Vpif_RawCapturePinPol                      capFieldPol;
    Vpif_RawCapturePinPol                      capVPixPol;
    Vpif_RawCapturePinPol                      capHPixPol;
}Vpif_CapChanParams;
```

Fields

`capStdMode` Operation mode title.

`capChannelIoMode` Operation mode for which the channel is opened

`capFbParams` Frame buffer settings.

`capStorageMode` Indicates whether it is field or frame based storage mode.

`*capEdcTbl` Function table of decoder module for the channel. A statically defined EDC function table is passed to the vpifMdCreateChan() function via the channel parameters argument.

`*capVideoParams` Specify the Video parameters if application would like to specify them. This is an optional parameter. If not used, set this element to NULL. If set to NULL, the driver will read the parameters depending upon the mode set. If it is not NULL these values will prevail over whatever mode being set. CAUTION: If wrong parameters are sent, the driver does not verify the validity of these parameters.

`capVbiService` Indicates what type VBI services are required by this mode. Available values for this field are defined in "*Vpif.h*" file with VPIF VBI Ancillary Data service title.

`capVbiSliceService` If the VBI type is Slice VBI then what kind of service it is. Valid only if one of the "*vbiService*" is set as *Vpif_VbiServiceType_SLICE_VBI*. Whatever slice service is set here only that data is captured. Available values for this field are defined in "*Vpif.h*" file with VPIF Slice VBI service type title.

`capDataSize` The data width bit is only used with the CCD/CMOS data capture mode.

`capFieldPol` Field ID polarity inverting control.

`capVPixPol` Vertical pixel valid signal polarity control.

`capHPixPol` Horizontal pixel valid signal polarity control.

Comments

"*capDataSize*", "*capFieldPol*", "*capVPixPol*" and "*capHPixPol*" are only for RAW capture mode they are not valid for BT mode.

Constraints

None

See Also

Vpif_RawCapturePinPol, Vpif_IoMode, Vpif_FrameBufferParams, Vpif_VideoMode, Vpif_SdramStorage

4.2.6 Vpif_HwInfo

This structure store the hardware related information of VPIF.

Definition

```
typedef struct Vpif_HwInfo_t
{
    CSL_VpifRegsOvly    baseAddr;
    Uint16              cpuEventNumber;
    Uint16              hwiNo;
    Uint32              pwrmLpscID;
    Uint32              pscInstance;
}Vpif_HwInfo;
```

Fields

baseAddr	Base address of VPIF hardware.
cpuEventNumber	Cpu interrupt number with which this device is associated.
hwiNo	HWI number for the ECM group in which the event is configured.
pwrmLpscID	LPSC Id of VPIF
pscInstance	PSC instance number for VPIF

Comments

"*hwiNo*" needs to be specified according to the ECM event group that the Vpif being configured falls in to.

Constraints

None

See Also

CSL_VpifRegsOvly

4.2.7 Vpif_PwrmInfo

This structure store the PWRM related information for VPIF.

Definition

```
typedef struct Vpif_PwrmInfo_t
{
    Vpif_PllDomain          pllDomain;
    PWRM_Event              pwrmEvent;
    PWRM_NotifyHandle       notifyHandle[Vpif_MAX_PWRM_EVENTS];
    Fxn                     delayedCompletionFxn[Vpif_MAX_PWRM_EVENTS];
}Vpif_PwrmInfo;
```

Fields

`pllDomain`	PLL domain to be used for the device.
`pwrmEvent`	Current PWRM event being processed.
`notifyHandle`	Handles required for unregistering of the events with PWRM.
`delayedCompletionFxn`	Delayed completion callback function pointer

Comments

None

Constraints

None

See Also

`PWRM_NotifyHandle, PWRM_Event, Vpif_PllDomain`

4.2.8 Vpif_Module_State

This tells the state of the instance.

Definition

```
typedef struct Vpif_Module_State_t
{
    Bool *inUse;
}Vpif_Module_State;
```

Fields

`*inUse`	Maintain use state of each Vpif instance.

Comments

None

Constraints

None

See Also

None

4.3 API Definition

4.3.1 Vpif_init

Syntax

```
Void Vpif_init(Void);
```

Arguments

IN	Void
	No Input Arguments

Return Value

Void	No return value.

Comments

Function to initialize the VPIF driver data structures.

Constraints

- This function should be called by the application before creating the driver instance.
- This function should be called only once in the life time of the Application.

See Also

None

4.3.2 vpifMdBindDev

Syntax

```
static Int32 vpifMdBindDev(Ptr *devp, Int32 devId, Ptr devParams);
```

Arguments

OUT	Ptr	*devp
	Pointer to Vpif driver object. Pointer to be updated once instance is created.	
IN	Int32	devId
	Number of device instance.	
IN	Ptr	devParams
	Device Parameters	

Return Value

IOM_COMPLETED	This component has been successfully initialized.
IOM_EBADARGS	Not valid arguments.
IOM_EBADMODE	Bad mode.

Comments

This function would be implemented at the IOM layer. This function would create instance of the driver and initialize the driver as well.

Constraints

- The Driver supports only one instance so "*devId*" should not be more than 1.
- The driver should be opening for the first time at the OS initialization time.

See Also

`Vpif_Object, Vpif_Params`

4.3.3 vpifMdUnBindDev

Syntax

```
static Int32 vpifMdUnBindDev(Ptr devp);
```

Arguments

```
IN       Ptr                          devp
```

Pointer to Vpif driver object.

Return Value

IOM_COMPLETED	This component has been successfully de-initialized.
IOM_EBADARGS	Not valid arguments.

Comments

"*devp*" must be a valid pointer and should not be null.

Constraints

None

See Also

`Vpif_Object`

4.3.4 vpifMdCreateChan

Syntax

```
static Int32 vpifMdCreateChan(Ptr              *chanp,
                              Ptr              devp,
                              String           name,
                              Int32            mode,
                              Ptr              chanParams,
                              IOM_TiomCallback cbFxn,
                              Ptr              cbArg);
```

Arguments

```
OUT      Ptr *                        chanp
```

Void pointer to be updated after the channel has been initialized. This is the channel object pointer used for further transaction on the channel.

```
IN       Ptr                          devp
```

Pointer to the Vpif instance structure

IN	String	name
	Vpif Instance name like Vpif0. Please see the naming format to pass this argument correctly.	
IN	Int32	mode
	Channel mode of the driver i.e. INPUT or OUTPUT.	
IN	Ptr	chanParams
	Channel parameters from user needed to initialize the channel.	
IN	IOM_TiomCallback	cbFxn
	Callback function pointer to the GIO	
IN	Ptr	cbArg
	Callback function arguments	

Return Value

IOM_COMPLETED	This channel has been successfully opened.
IOM_EBADARGS	Bad arguments are passed.
IOM_EBADIO	Bad IO happened because of EDC open

Comments

This function creates a communication channel in specified mode to communicate data between the application and the Vpif channel. This function sets the required hardware configurations for the data transactions. It returns configured channel handle to application, which will be used in all further transactions with the channel. This function is called in response to FVID_create()/GIO_create() call.

Constraints

- The instance of driver should be created and be initialized.
- The channel should be closed before opening it.
- For capture "*mode*" is required to be passed as *IOM_INPUT* and for display *IOM_OUTPUT*.
- Channel 0 and 1 must be used for capture and channel 2 and 3 must be used for display.

See Also

Vpif_Object, Vpif_ChanObj, Vpif_CapChanParams, Vpif_DisChanParams

4.3.5 vpifMdDeleteChan

Syntax

```
static Int32 vpifMdDeleteChan(Ptr chanp);
```

Arguments

`IN`	`Ptr`	`chanp`
	Pointer to channel object returned during channel creation.	

Return Value

`IOM_COMPLETED`	The channel has been successfully deleted.
`IOM_EBADMODE`	Channel state is not "Opened"
`IOM_EBADARGS`	Bad arguments are passed.
`IOM_EBADIO`	EDC close result in error.

Comments

This function is called by the application to close a previously opened channel. It deletes the channel so that it is not available for further transactions. All the allocated resources are freed & the channel will be ready for the "open" operation once again.

Constraints

Channel should be in open state before calling this function.

See Also

`Vpif_ChanObj, Vpif_Object`

4.3.6 vpifMdSubmitChan

Syntax

```
static Int32 vpifMdSubmitChan(Ptr chanp, IOM_Packet *ioPacket);
```

Arguments

`IN`	`Ptr`	`chanp`
	Pointer to channel object returned during channel creation.	
`IN OUT`	`IOM_Packet *`	`ioPacket`
	IO packet pointer	

Return Value

`IOM_COMPLETED`	If packet is fully processed.
`IOM_PENDING`	If packet is not fully processed.
`Negative value`	In case of error.

Comments

This function submits I/O packet to a channel for processing. The FVID/GIO layer calls this function to cause the mini-driver to process the *`IOM_Packet`* for read/write operations. This function is called to Queue/Dequeue Frame buffer.

Constraints

None

See Also

`Vpif_ChanObj`

4.3.7 vpifMdControlChan

Syntax

```
static Int32 vpifMdControlChan(Ptr chanp, Uns cmd, Ptr cmdArg);
```

Arguments

IN	Ptr	chanp
	Pointer to channel object returned during channel creation.	
IN	Uns	cmd
	IOCTL command code.	
IN OUT	Ptr	cmdArg
	Optional argument required for command execution.	

Return Value

IOM_COMPLETED	The IOCTL is successfully processed.
IOM_EBADARGS	Wrong arguments passed
IOM_EBADMODE	The driver state is not appropriate.
IOM_EBADIO	IO related error
Others	In case of error.

Comments

This function executes a control command passed by the application. The application's request for IOCTL to be executed is routed here by the stream. If the command is supported then the control command is executed.

Constraints

None

See Also

`Vpif_ChanObj`

5 Decision Analysis & Resolution

5.1 Driver Design

5.2 Available Alternatives

5.2.1 Alternative 1

Develop a completely new design for VPIF driver from scratch.

5.2.2 Alternative 2

Use the FVID based driver design from DM648.

5.3 Decision

Considering the timing constraints BU decided to go ahead with the Alternative 2.

5.4 Addressing the channel

Addressing the channel and string parameter passing for EDC driver

5.5 Available Alternatives

5.5.1 Alternative 1

Pass the sting as "`/VPIF0/A/0x5C`"

5.5.2 Alternative 2

Use "`/VPIF0/0/I2C0/TVP5147_0/0x5C`"

5.6 Decision

It is suggested to use 0 instead of A, as it will conform to the VPIF channel numbers. It is also decided to use I2C0 also as a part of string in order to make the driver/application extensible. If SPI comes up for programming the EDC drivers it can be easily plugged in. Also in order to not hard code the I2C instance number, it should be passed as a parameter which instance is used. It is also suggested to use the EDC device name also as a part of this string. This can be useful if two decoders are connected and one of them is required to be bypassed like: Ch A – MUX - TVP1 – TVP2. So it was decided to go ahead with alternative 2.

5.7 Codec Interface layer

Codec Interface layer should be used to program the H/W codec. The EDC drivers should not directly program the encoder and decoder.

5.8 Available Alternatives

5.8.1 Alternative 1

Do not take parameter for the type of interface and assume I2C as the connection mode.

5.8.2 Alternative 2

Take parameter about the type of interface.

5.9 Decision

It is decided to use alternative 2 such that it can be extensible to SPI mode of control in future.

5.10 CSL Discussion

5.11 Available Alternatives

5.11.1 Alternative 1

Use the register overlays as is provided by VPIF hardware

5.11.2 Alternative 2

Club the capture and display channel separately.

5.12 Decision

Alternative 1 for capture suggests following piece of code:

```
if (channelNo == 0)

{

    channel->baseAddress->C0TLUMA = topFieldY;

}

if (channelNo == 1)

{

    channel->baseAddress->C1TLUMA = topFieldY;

}
```

Alternative 2 for capture suggests following piece of code:

```
if (channelNo is capture (0 or 1))

{

    channel->baseAddress->CaptureRegs[channelNo]->TLUMA =
topFieldY;

}
```

It is decided to use alternative 2 to have less conditional checks in the driver.

5.13 Slice VBI Handling

Handling the Slice VBI data during VPIF capture and display

5.14 Available Alternatives

5.14.1 Alternative 1

Once an ISR comes, for interlaced mode (field 0) SWI is posted. In SWI the VPIF interrupts are disabled and then I2C read/write (blocking) happens from ADV7343 or TVP5147. For field 0, current frame is processed hereafter SWI processing i.e. if an application request is pending then the frame is released. If not then put it in out Q for display or make this as most recent IOP for capture. If the interrupt is because of filed 1, then again SWI post happens. In SWI the VPIF interrupts are disabled and a read/write happens. Once this SWI completes VPIF registers are updated with the new frame pointers.

5.14.2 Alternative 2

VPIF driver do not worry about the Slice VBI data, as it is not related. Application when wants the VBI data, calls the ADV7343 or TVP5147 directly

5.14.3 Alternative 3

From the ISR SWI 0 is posted for field 0. SWI 0 calls the I2C read/write and gives a callback function to the TVP1547/ADV7343. This callback function will be called once i2c transfer completes and then the frame is processed i.e. if an application request is pending the frame is released here. If not then put it in out queue for display or make this as most recent IOP for capture. For field 1 SWI 1 is posted which read/write field 1 data. From ISR once the SWI 1 is posted then update the VPIF registers with the new frame pointers.

5.14.4 Alternative 4

It’s a variation of alternative 3 where SWI is not used. Issues I2C read/write from the ISR itself, with callback function and everything else as above.

5.14.5 Alternative 5

It’s another variation of alternative 4. Here when the callback comes SWI is issued where all the processing happens.

5.15 Decision

For alternative 1 the problem is

- From SWI i2c read/write calls cannot be sync calls, as Sem_pend() cannot be done from SWI.
- Wait for SWI completion to start next transfer.

Alternative two is simpler from implementation perspective for driver but has limitation where

- Application will be loaded as it needs to call read/write VBI data for every video frame through TVP5147/ADV7343 IOCTL.
- Application needs to be aware for TVP5147 and ADV7343 calls and VBI data is not a part of the video frame.

Alternative design 3 does not require wait for SWI to complete but has limitation where

- May lose in between SWI’s.

- Programming is complex.

For alternative 4 in-between SWI's will not be lost but has limitation where

- I2C callback is too bulky. Increases the I2C ISR time.
- Programming is complex.

Alternative 5 does not make I2C Callback bulky but the programming is complex.

Considering all the alternatives and their pros and cons, it is decided to use two task, one for capture and one for display, where the slice VBI data is read and written. The write calls to the ADV7343 are async calls and does not block the task for display. The read calls to TVP5147 are sync calls, as a read is composed of one write and one read to the I2C slave (TVP5147) for capture. Only semaphores are posted from the ISR, which does not block the ISR.

For capture, blocking read call takes place from the task and once it is done the capture packet processing takes place.

For display, non blocking write call takes place from the task and the call returns immediately. In the callback of this write call only SWI is posted, so that the I2C callback is not bulky. From the SWI display packet processing takes place.

6 Revision History

Version #	Date	Author Name	Revision History
00.10	03 Aug 2009	Vipin Bhandari	Document created
00.20	17 Sep 2009	Vipin Bhandari	Review comments addressed
01.00	13 Nov 2009	Vipin Bhandari	Slice VBI handling added

««« § »»»