HDVPSS DRIVER
USER GUIDE

Contents

Articles

References

TI81xx-HDVPSS-UserGuide

About this Manual

This document describes how to install and work with the Texas Instruments TI81xx HDVPSS drivers on TI81xx EVM. The HDVPSS package serves to provide a fundamental software platform for development, deployment and execution of video applications. HDVPSS abstracts the functionality provided by the hardware and forms the basis for all video applications development on this platform.

In this context, the document contains instructions to:

- Install the HDVPSS package
- Build the HDVPSS package

The document provides overview of HDVPSS and the following drivers contained in HDVPSS package:

- HDVPSS introduction
- HDVPSS Display drivers
- HDVPSS Memory to Memory drivers
- HDVPSS Capture drivers

Please note that ES 1 refer to PG 1.0 Silicon, ES 1.1 refer to PG 1.1 silicon, ES 2.0 refer to PG 2.0 silicon, ES 2.1 refer to PG 2.1 silicon. If no ES qualifier is used, it is applicable to all the known silicon versions.

Getting Started

System Requirements

Refer to 'Dependencies' section of release notes.

Installation

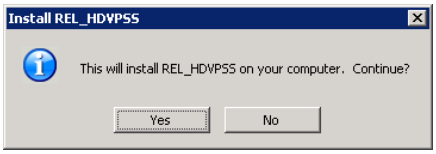
HDVPSS device driver installation package is a self extracting EXE. Double click the installation package and install the package in the directory where various Texas Instruments tools are installed like CCS, BIOS, etc.

Following are the installation steps:

- Double click the installer package. Language screen appears. Select the language and click OK.

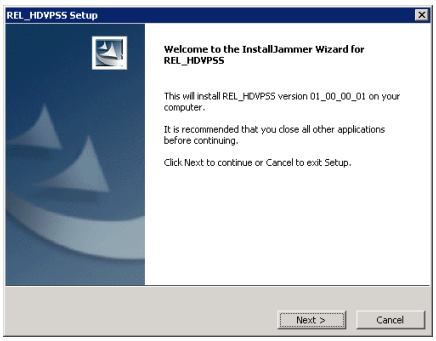
Installation step 1

- Dialogue box appears to confirm the package installation click Yes to install.



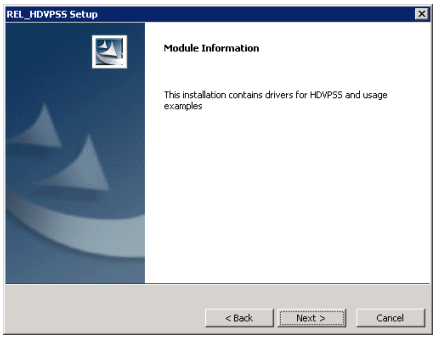
Installation step 2

- Welcome screen appears. Click next to continue.



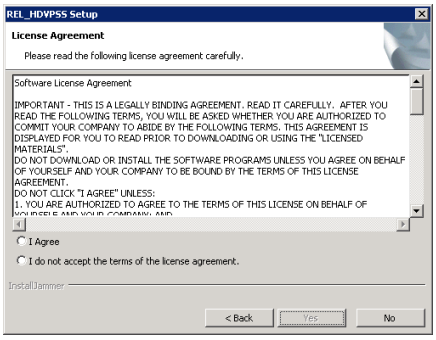
Installation step 3

- Module information appears for the installation package. Click next to continue.



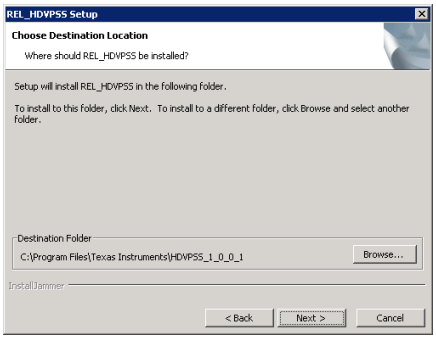
Installation step 4

- License agreement appears. Accept the terms of agreement after reading and click yes to continue.



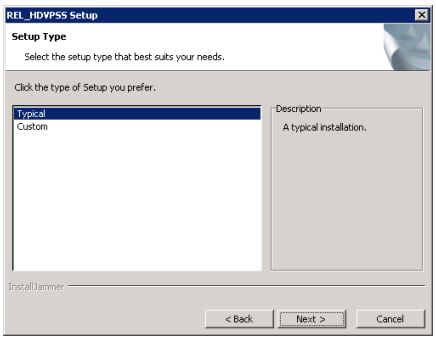
Installation step 5

- Destination folder screen appears. Select the installation folder where other TI tools like CCS, BIOS are installed.



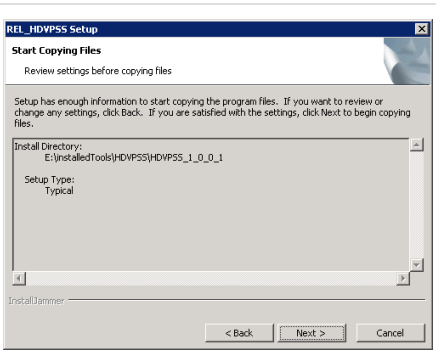
Installation step 6

- Select Typical type of installation.



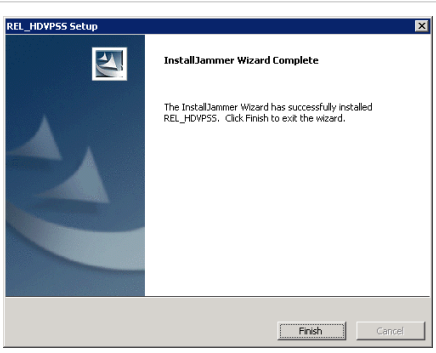
Installation step 7

- Confirmation screen appears. Click next to install the package.



Installation step 8

- Click on Finish to complete the installation



Installation step 9

This completes the installation of the package.

Running First HDVPSS Application on TI816x

This section describes the out of the box experience for the HDVPSS drivers. It shows how to run the first HDVPSS application involving major HDVPSS drivers.

Demo application on Video Surveillance (VS) Application board

Following are the application details which is running on the VS application board:

- 16 channels NTSC interlaced input to VIP capture driver using TVP5158 in line multiplexed mode.
- VIP capture outputs all the buffers in the memory in YUV422 interlaced format.
- DEI high quality and DEI medium quality picks 8 channels each out of 16 channels and de-interlaces the interlaced image and scales it to CIF size and outputs it to memory.
- Display operates in multi window mode of 4X4 window picking up one buffer from each of the 16 channels and displays it in mosaic fashion at 1080P30 resolution.

More about the integration application is explained at Integration Application

Steps to run the Demo on Video Surveillance Board

Common Steps for all examples

- Switch on the board.
- Open the CCS and connect to TI816x (CortexA8) using the CCS and debugger.
- Load gel file ***TI816x_evm_A8_ddr3.gel*** under directory *$HDVPSS_INSTALL_DIR/docs/TI816x* for A8 processor.
- Run *Scripts* > *TI816x HDVPSS Init* > *HDVPSSInit*. This will enable the DDR, Ducati and HDMI.
- Connect to CortexM3_ISS
- Load gel file ***TI816x_evm_ducati.gel*** file in ISS_M3 processor under directory *$HDVPSS_INSTALL_DIR/docs/TI816x*.
- Run *scripts* > *UnicacheEnableDisable* > *Ducati_cache_enable*. This enables the cache on Ducati.

Steps for specific application

- Connect 16 input sources to the composite input of the video surveillance board.
- Connect the HDMI output of the VS board to the HDMI input of the TV.
- Load and Run $HDVPSS_INSTALL_DIR/pspdrivers_/build/hdvpss_examples_chains/bin/ti816x-evm/hdvpss_examples_chains_m3vpss_debug
- Select option 7 at " Enter Choice:" option.
- You can see the 4X4 window of the 16 captured channels.

Important

Currently frame drops is observed on all the channels because of the DDR2 bandwidth limitation and/or the application issue.

Running First HDVPSS Application TI814x

This section describes the out of the box experience for the HDVPSS drivers. It shows how to run the first HDVPSS application involving major HDVPSS drivers.

Demo application on Video Surveillance (VS) Application board

Following are the application details which is running on the VS application board:

- 4 channels NTSC interlaced input to VIP capture driver using TVP5158 in pixel multiplexed mode.
- VIP capture outputs all the buffers in the memory in YUV422 interlaced format.
- DEI de-interlaces the interlaced image and scales it to CIF size and outputs it to memory.
- Scaler scales the images to fit into 1080P window
- Display operates in single window mode and display at 1080P30 resolution.

More about the integration application is explained at Integration Application

Steps to run the Demo on Video Surveillance Board

Common Steps for all examples

- Switch on the board.
- Open the CCS and connect to TI814x (CortexA8) using the CCS and debugger.
- Load gel file ***TI814x_ES_xx_evm_A8_ddrX.gel*** under directory *$HDVPSS_INSTALL_DIR/docs/TI814x* for A8 processor.
- Use TI814x_ES_1_evm_A8_ddr2.gel for ES 1 DDR 2 board
- Use TI814x_ES_2x_evm_A8_ddr2.gel for ES 2.1 DDR 2 board
- Use TI814x_ES_2x_evm_A8_ddr3.gel for ES 2.1 DDR 3 board
- Run *Scripts > TI814x HDVPSS Init > HDVPSSInit*. This will enable the DDR, Ducati and HDMI.
- Connect to CortexM3_ISS
- Load gel file ***TI814x_evm_ducati.gel*** file in ISS_M3 processor under directory *$HDVPSS_INSTALL_DIR/docs/TI814x*.
- Run *scripts > UnicacheEnableDisable > Ducati_cache_enable*. This enables the cache on Ducati.

Steps for specific application

- Connect 8 input sources to the composite input of the video surveillance board.
- Connect the HDMI output of the VS board to the HDMI input of the TV.
- Load and Run $HDVPSS_INSTALL_DIR/pspdrivers_/build/hdvpss_examples_chains/bin/ti814x-evm/hdvpss_examples_chains_m3vpss_debug
- Select option 4 at " Enter Choice:" option.
- You can see the 2X2 window of the 4 captured channels.

Running First HDVPSS Application on TI8107

This section describes the out of the box experience for the HDVPSS drivers. It shows how to run the first HDVPSS application involving major HDVPSS drivers.

Demo application on Video Surveillance (VS) Application board

Following are the application details which is running on the VS application board:

- 4 channels NTSC interlaced input to VIP capture driver using TVP5158 in pixel multiplexed mode.
- VIP capture outputs all the buffers in the memory in YUV422 interlaced format.
- DEI de-interlaces the interlaced image and scales it to CIF size and outputs it to memory.
- Scaler scales the images to fit into 1080P window
- Display operates in single window mode and display at 1080P60 resolution.

More about the integration application is explained at Integration Application

Steps to run the Demo on Video Surveillance Board

Common Steps for all examples

- Switch on the board.
- Open the CCS and connect to CortexA8 using the CCS and debugger.
- Load gel file ***TI8107_ES_1_evm_A8_ddr3.gel*** under directory *$HDVPSS_INSTALL_DIR/docs/TI8107* for A8 processor.
- Run *Scripts* > *TI8107 System Initialization* > *TI8107HdvpssInit*. This will enable the DDR, Ducati and HDMI.
- Run *Scripts* > *TI8107 VDB DDR Configurations* > *EVM_DDR3_EMIF0_400MHz_Config_256MB*. This will enable DDR for 256MB size. Please note that this option is required only for NetCam card.
- Connect to CortexM3_ISS
- Load gel file ***TI8107_evm_ducati.gel*** file in ISS_M3 processor under directory *$HDVPSS_INSTALL_DIR/docs/TI8107*.
- Run *scripts* > *UnicacheEnableDisable* > *Ducati_cache_enable*. This enables the cache on Ducati.

Steps for specific application

- Connect 4 input sources to the composite input of the video surveillance board.
- Connect the HDMI output of the VS board to the HDMI input of the TV.
- Load and Run $HDVPSS_INSTALL_DIR/pspdrivers_/build/hdvpss_examples_chains/bin/ti8107-evm/hdvpss_examples_chains_m3vpss_debug
- Select option 5 at " Enter Choice:" option.
- You can see the 2X2 window of the 4 captured channels.

Compiling HDVPSS Drivers

HDVPSS drivers and examples can be built using the *Makefile* present in the HDVPSS installation directory. Before doing so, user may have to modify the *Rules.make* file (present in the installation directory), depending upon his build environment.

Open the *Rules.make* file and make sure that the paths for the following tool-chains are correct (default installation paths for all the required tool-chains can be found in this file):

- **CODEGEN_PATH_M3** := Points to the Codegen toolchain for M3.
- **hdvpss_PATH** := Points to the HDVPSS installation directory.
- **bios_PATH** := Points to the BIOS installation directory.
- **xdc_PATH** := Points to the XDC installation directory.
- **ipc_PATH** := Points to the IPC installation directory.

Important

Make sure that the above mentioned paths don't have white spaces in them. In case the installation directory has any white space, use the short names generated by issuing `dir /X` on the DOS command prompt, which replaces the white spaces by ~. Also, use forward slash '/' instead of back slash '\' in all the above paths.

To build on Linux, an additional step is required to set the `OS` environment variable to `Linux`. Alternatively, it can be passed to the gmake command as indicated in the below steps:

After editing the *Rules.make* file, open a command prompt and `cd` (change directory) to the HDVPSS installation directory.

```
>cd $(HDVPSS_INSTALL_DIR)
```

Provide the command `gmake -s all`. This will clean and recursively build all the libraries and examples for the default platform (ti816x-evm) and default profile (whole_program_debug).

```
>gmake -s all
```

To build on Linux, if the `OS` environment variable is not set, the gmake command can be invoked as:

```
>gmake -s all OS=Linux
```

If the command prompt can't locate *gmake* command, then add the directory where *gmake* is present in the PATH environmental variable. Typically, *gmake* comes along with CCS/XDC installation and can be found at *$(CCS_INSTALL_DIR)/xdctools_XX_YY_ZZ_WW*.

Note Default platform and profile can be changed by modifying `PLATFORM` and `PROFILE_$(CORE)` in the *Rules.make* file respectively.

During development, the below *gmake* targets can also be used be used for convenience:

- **gmake -s hdvpss** - incrementally builds only HDVPSS drivers
- **gmake -s examples** - incrementally builds HDVPSS drivers and all examples
- **gmake -s examples_netcam** - incrementally builds HDVPSS drivers and all examples for the netcam/vcam usecase
- **gmake -s clean** - clean all drivers and examples
- **gmake -s examplesclean** - clean all examples ONLY
- **gmake -s *example_name*** - incrementally builds HDVPSS drivers and the specific example ONLY. Values for *example_name* can be - i2c, captureVip, chains, display etc.

Important
The detailed build instructions with all the supported options can be found in the README.txt file in the HDVPSS installation directory.

Memory Map for TI816x

The following shows the memory map of the DDR/OCMC memory used for the various sections and for the different processors in the system. This assumes a 1GB DDR memory and 512KB OCMC memory. The Video/DSS M3 MMU is configured such away that the 1GB DDR is split into two sections - 1st 512MB is cached and the next 512MB is non-cached.

- 1st 512MB - Cached:
  - Linux: Used by the linux kernel running from A8. This is not used by M3.
  - Tiler 16-bit, CMEM, DSP: These are not used by M3.
  - Syslink IPC [SR0, SR1]: Shared memory used by syslink module to perform inter processor communication.
  - DSS M3 Code: DSS M3 code program section (driver text section resides here)
  - Video M3 Code: Video M3 code program section
  - Video M3 Data: Video M3 bss and other data section
  - DSS M3 Data: DSS M3 bss and other data section (driver bss, const and other global variables reside here)
  - SHARED CTRL DUCATI, Shared Data, Debug: These are not used by M3.
- 2nd 512MB - Non-Cached:
  - Notify Mem: Used as notify shared memory
  - HDVPSS Shared Mem: Shared memory used for IPC between HDVPSS proxy server and client on A8 like FBDEV driver.
  - VPDMA Desc Mem: HDVPSS driver VPDMA descriptor memory section used to store descriptors and overlay memory
  - Frame Buffer: Frame buffer heap used for non-tiled video buffers
  - Tiler 8-bit/Tiler page: Tiler memory for the different tiler view

```
DDR: 0x80000000 (Ist 512MB - Cached)
+-----------------+
```

```
|                 |
| Linux           | 256MB
|                 |
+-----------------+
| Tiler 16-bit    | 64MB
+-----------------+
| CMEM            | 10MB
+-----------------+
| DSP             | 32MB
+-----------------+
| IPC (SR1)       | 12MB
+-----------------+
| IPC (SR0)       | 16MB
+-----------------+
| DSS M3 Code     | 4MB
+-----------------+
| Video M3 Code   | 4MB
+-----------------+
| Video M3 Data   | 32MB
+-----------------+
| DSS M3 Data     | 60MB
+-----------------+
| SHARED CTRL     |
| DUCATI          | 11MB
+-----------------+
| Shared Data     | 1MB
+-----------------+
| Debug/NOT USED  | 10MB
+-----------------+

DDR: 0xA0000000 (2nd 512MB - Non-Cached)
+-----------------+
| Notify Mem      | 2MB
+-----------------+
| HDVPSS Shared   | 3MB
| Mem             |
+-----------------+
| VPDMA Desc Mem  | 3MB
+-----------------+
| FrameBuffer     | 248MB
+-----------------+
| Tiler PAGE      | 128MB
|                 |
+-----------------+
| Tiler 8-bit     | 128MB
|                 |
+-----------------+
```

```
OCMC: 0x40300000
+-----------------+
| OCMC0           | 256KB
+-----------------+

OCMC: 0x40400000
+-----------------+
| OCMC1           | 256KB
+-----------------+
```

Memory Map for TI814x

The following shows various sections defined and used by HDVPSS drivers and its sample applications. This assumes a 512 MB of DDR memory and 128KB OCMC memory. The Video/DSS M3 MMU is configured such away that the 512MB DDR is split into two sections - 1st 256MB is cached and the next 256MB is non-cached.

- 1st 256MB - Cached:
  - Linux: Used by the linux kernel running from A8. This is not used by M3.
  - Sections EVENT_LIST_CORE0, PRIVATE_CORE0_DAT and EXTMEM_CORE0 is not used by M3 HDVPSS
  - Syslink IPC [SR0]: Shared memory used by syslink module to perform inter processor communication.
  - VPSS M3 Data: VPSS M3 bss and other data section (driver bss, const and other global variables reside here)
  - VPSS M3 Code: VPSS M3 code program section (driver text section resides here)
  - Debug: Not used
- 2nd 256MB - Non-Cached:
  - Notify Mem: Used as notify shared memory
  - Tiler 8-bit/16-bit: Tiler memory for the different tiler view
  - Frame Buffer: Frame buffer heap used for non-tiled video buffers
  - VPDMA Desc Mem: HDVPSS driver VPDMA descriptor memory section used to store descriptors and overlay memory
  - HDVPSS Shared Mem: Shared memory used for IPC between HDVPSS proxy server and client on A8 like FBDEV driver.

```
DDR: 0x80000000 (Ist 256MB - Cached)
+-----------------+
|                 |
| Linux           | 83MB
|                 |
+-----------------+
| EVENT_LIST_CORE0| 10MB
+-----------------+
|PRIVATE_CORE0_DAT| 37MB
+-----------------+
| EXTMEM_CORE0    | 0.625MB - or 625KB
+-----------------+
| Syslink IPC     | 16MB - SHARED_CTRL
+-----------------+
|                 |
```

```
| VPSS M3 Data    | 53MB
|                 |
+-----------------+
| VPSS M3 Code    | 53MB
+-----------------+
| Debug/NOT USED  | 3MB
+-----------------+

DDR: 0xC0000000 (2nd 256MB - Non-Cached)

+-----------------+
|                 |
|   Tiler         | 128MB
+-----------------+
|                 |
| Frame Buffer    | 123MB
+-----------------+
| Notify Shared   | 1MB
|      Mem        |
+-----------------+
| VPDMA Desc Mem  | 2MB
+-----------------+
| HDVPSS Shared   | 2MB
|      Mem        |
+-----------------+



OCMC: 0x00300000
+-----------------+
| OCMC0 (Not used)| 128KB
+-----------------+
```

Memory Map for TI8107

The following shows various sections defined and used by HDVPSS drivers and its sample applications. This assumes a 512 MB of DDR memory and 256KB OCMC memory. The Video/DSS M3 MMU is configured such away that the 512MB DDR is split into two sections - 1st 256MB is cached and the next 256MB is non-cached.

- 1st 256MB - Cached:
  - Linux: Used by the linux kernel running from A8. This is not used by M3.
  - Syslink IPC [SR0]: Shared memory used by syslink module to perform inter processor communication.
  - VPSS M3 Data: VPSS M3 bss and other data section (driver bss, const and other global variables reside here)
  - VPSS M3 Code: VPSS M3 code program section (driver text section resides here)
  - Debug: Not used
- 2nd 256MB - Non-Cached:
  - Notify Mem: Used as notify shared memory
  - Tiler 8-bit/16-bit: Tiler memory for the different tiler view

- Frame Buffer: Frame buffer heap used for non-tiled video buffers
- VPDMA Desc Mem: HDVPSS driver VPDMA descriptor memory section used to store descriptors and overlay memory
- HDVPSS Shared Mem: Shared memory used for IPC between HDVPSS proxy server and client on A8 like FBDEV driver.

```
DDR: 0x80000000 (Ist 256MB - Cached)
+-----------------+
|                 |
| Linux           | 130.625MB
|                 |
+-----------------+
| Syslink IPC     | 16MB - SHARED_CTRL
+-----------------+
|                 |
| DSS M3 Data     | 53MB
|                 |
+-----------------+
| DSS M3 Code     | 53MB
+-----------------+
| Video M3 Data   | 1MB
+-----------------+
| Video M3 Code   | 1MB
+-----------------+
| Debug/NOT USED  | 1MB
+-----------------+


DDR: 0xA0000000 (2nd 256MB - Non-Cached)

+-----------------+
|                 |
|   Tiler         | 128MB
+-----------------+
|                 |
| Frame Buffer    | 123MB
+-----------------+
| Notify Shared   | 1MB
|      Mem        |
+-----------------+
| VPDMA Desc Mem  | 2MB
+-----------------+
| HDVPSS Shared   | 2MB
|      Mem        |
+-----------------+


OCMC: 0x00300000
```

```
+-----------------+
| OCMC0 (Not used)| 256KB
+-----------------+
```

Directory Organization

The following expands on the directory structure of → HDVPSS Folders

HDVPSS Overview

This section provides top level information about HDVPSS hardware architecture and software architecture.

HDVPSS Hardware Overview

HDVPSS hardware overview explains the HDVPSS hardware blocks in brief. It is not necessary to have a full knowledge of the HDVPSS hardware architecture to use HDVPSS drivers. HDVPSS hardware overview can be found at HDVPSS Hardware Overview.

HDVPSS Software Overview

HDVPSS software overview explains the HDVPSS software and the major class of drivers supported by the HDVPSS software interfaces to the application. It is not important to have a full understanding of the HDVPSS software architecture for using HDVPSS drivers. HDVPSS Software overview could be found at HDVPSS Software Overview

FVID2 Overview

FVID2 are the set of APIs or framework specifically designed for the video class of devices. It exposes number of features of the video devices in a standard way so that application needs a minimum changed while migrating from the once class of video devices to other class of video devices, both of them adhering to the FVID2 interfaces. More details about the FVID2 can be found at → UserGuideFVID2

HDVPSS Drivers

This section explains about the different class of the HDVPSS drivers supported by the HDVPSS package.

Platform APIs and Drivers

This section describes about APIs and driver which are required to initialize and setup platform for HDVPSS drivers. These API and drivers doesn't fit into FVID2 drivers since they are very much dependent on SoC and board. Users needs to modify these APIs based on their boards. Description of platform APIs and Drivers could be found at → HDVPSS Platform APIs and Drivers

Display Drivers

Display drivers refers to the drivers which takes the input buffer from the memory and displays that buffer on the external device like TV, LCD etc. Details of the display drivers supported by the HDVPSS packages can be found at → HDVPSS Display Driver UserGuide

Memory Drivers

Memory drivers refers to the drivers which takes the input from the memory, processes the input like scale the image, chroma up samples the image and puts it back to the memory. Details of the memory drivers supported by the HDVPSS package can be found at → HDVPSS M2M Driver UserGuide

Capture Driver

Captures drivers refers to the drivers which takes the input from the external sources like camera, dvd players etc and puts the capture images in the memory. Details of the capture drivers supported by the HDVPSS package can be found at → HDVPSS Capture Driver UserGuide

External video device drivers

External video device drivers refers to the devices which are external to the HDVPSS like the TVP5158 decoder, sil9022a HDMI encoder etc. Details about the external video device drivers could be found at → External Video Device Drivers

Integration Examples

Integration examples demos the different drivers used in various combination. It shows some specific real world applications like 16 channel capture, noise filter it, de-interlace the interlaced capture and show it on the TV in mosaic format. Details of the different integration applications can be found at → HDVPSS Integration Examples

HDVPSS Software Support on MultiCore Architecture

HDVPSS drivers supports the multi core architecture. All the HDVPSS driver interfaces can be exposed on any other processor running different OS, using the HDVPSS multi core software architecture. One such use-case is controlling the graphics plane of the HDVPSS through the Linux Os running on the A8 processor. Details about the HDVPSS multi core architecture can be found at → HDVPSS Software MultiCore Architecture

HDVPSS Linux Drivers

Build ProxyServer BIOS Application

Installation

- Change the IPC path in Rules.make to the installed directory
- Run command prompt and cd (change directory) to the HDVPSS install directory

```
>cd $(HDVPSS_INSTALL)
```

- Provide the command *gmake -s proxy*. This will build proxyserver BIOS application, which is loaded by the syslink slaveloader user space application.

```
>gmake -s proxy
```

To build on Linux, if the *OS* environment variable is not set, the gmake command can be invoked as:

```
>gmake -s all OS=Linux
```

Proxy with Display App

Introduction This sample app is intended to run an HDVPSS application on M3 when there is one more application HDVPSS running on A8. This application runs the mosaic display test from Off-Chip HDMI on the platform. The user can run another linux application on A8 (for ex: fbdev application) which can output on On-Chip HDMI and check whether both the application can run in parallel.

Compilation To compile the application for linux type the following on command line

```
>gmake -s proxyDisplay OS=Linux
```

Running the application on the board To run this application the following are the steps to be done

- Boot up linux on the board
- Load Syslink module
- Load VPSS-M3 firmware (this module)
- load VPSS-M3 module
- Load fbdev module
- Load on-chip HDMI module

(For more information on how the above can be done refer to the section "Load VPSS and Fbdev Driver Modules" in PSP Video Driver User guide) When the above is done there will be output from the off-chip HDMI to display. Now the user can run another application on A8 to test whether the application on A8 and M3 can run in parallel (for ex: fbdev application which outputs to On-chip HDMI)

Linux FrameBuffer Driver

This section primarily describes about the drivers which are not the part of HDVPSS package but uses the HDVPSS drivers. Those drivers includes Linux framebuffer driver on Graphics pipeline, V4L2 display and capture driver on display and VIP pipelines respectively. Details about all those drivers can be found at http://processors.wiki.ti.com/index.php/TI81XX_PSP_User_Guide

HDMI Driver

HDMI driver is another driver which is controlled from the Linux running the on A8. Currently the HDMI driver interfaces are exposed as a part of the Linux standard Character driver interface. Details about the same can be found at HDMI Driver

T181xx-HDVPSS Overview

HDVPSS Hardware Introduction

TI814X/TI8107 HDVPSS Hardware

→ TI814X/TI8107 HDVPSS Hardware Overview

TI816X HDVPSS Hardware

→ TI816X HDVPSS Hardware Overview

HDVPSS Driver Introduction

HDVPSS drivers could be divided at top level in three categories:

- Display Driver
- Memory Driver
- Capture Driver

Display Driver

There are several display drivers path are possible going through DEI Aux and main,422BP path and Secondary path. Apart from this, there is support for three GRPX path which supports only RGB different data format. More details about display driver is discussed in respective section. Some of salient features are mentioned below:

- Display driver will support different standard resolution like 1080p60, 1080i60,720p60 etc for HD VENC and SD resolution for SD VENC.
- All Display drivers are non-blocking i.e. asynchronous drivers. Blocking calls are not supported.
- Notification of operation completion is done through callbacks.
- Different paths within display drivers could be configured using display controller.
- Once the display operation is started, the display driver always retains the last buffer and displays the same buffer continuously till the application gives a new buffer to display.
- Some of display driver will have capability for inline scaling and de-interlacing. These capabilities are based on whether specific IPs are available in the path or not.
- Display drivers will also be supported on the path having graphics IPs.
- All Display drivers including graphics are supported on:
    - Interfaces : FVID2
    - OS :BIOS6
    - Processor : Ducati M3

Memory (M2M) Driver

Like display driver, there are several paths which could be used for memory to memory operation. Some of standard M2M operations are like scaling, noise filtering, de-interlacing etc. Different supported M2M drivers are discussed in the respective section. Following are top level features of most of memory drivers:

- All Mem-Mem drivers are non-blocking i.e. asynchronous drivers. Blocking calls not supported!!
- M2M drivers could be opened multiple times – supports multiple handles (N) for the same driver - Each handle can have different configuration.
- Memory driver supports queuing of input request from several handles and calls returns immediately. Caller/Requester will be informed about completion of memory operation (Resizing/NF/ de-interlacing) through callbacks and there are different callbacks for each handles.
- Each call/requests to Mem Driver can consist of set of buffers.
- N (Maximum Number of picture or frame in each request) is fixed per handle at time of driver open.For example, its possible to submit a request 16CH buffer to NF and get only one notification at the end of 16CH NF processing. Thus reducing interrupts from 960 interrupts/sec to 60 interrupts/sec. This results in lower CPU load and higher HW utilization.
- Callback will be generated after processing all requests in a given set.
- All M2M drivers are supported on:
    - Interfaces : FVID2
    - OS :BIOS6
    - Processor : Ducati M3

Capture Driver

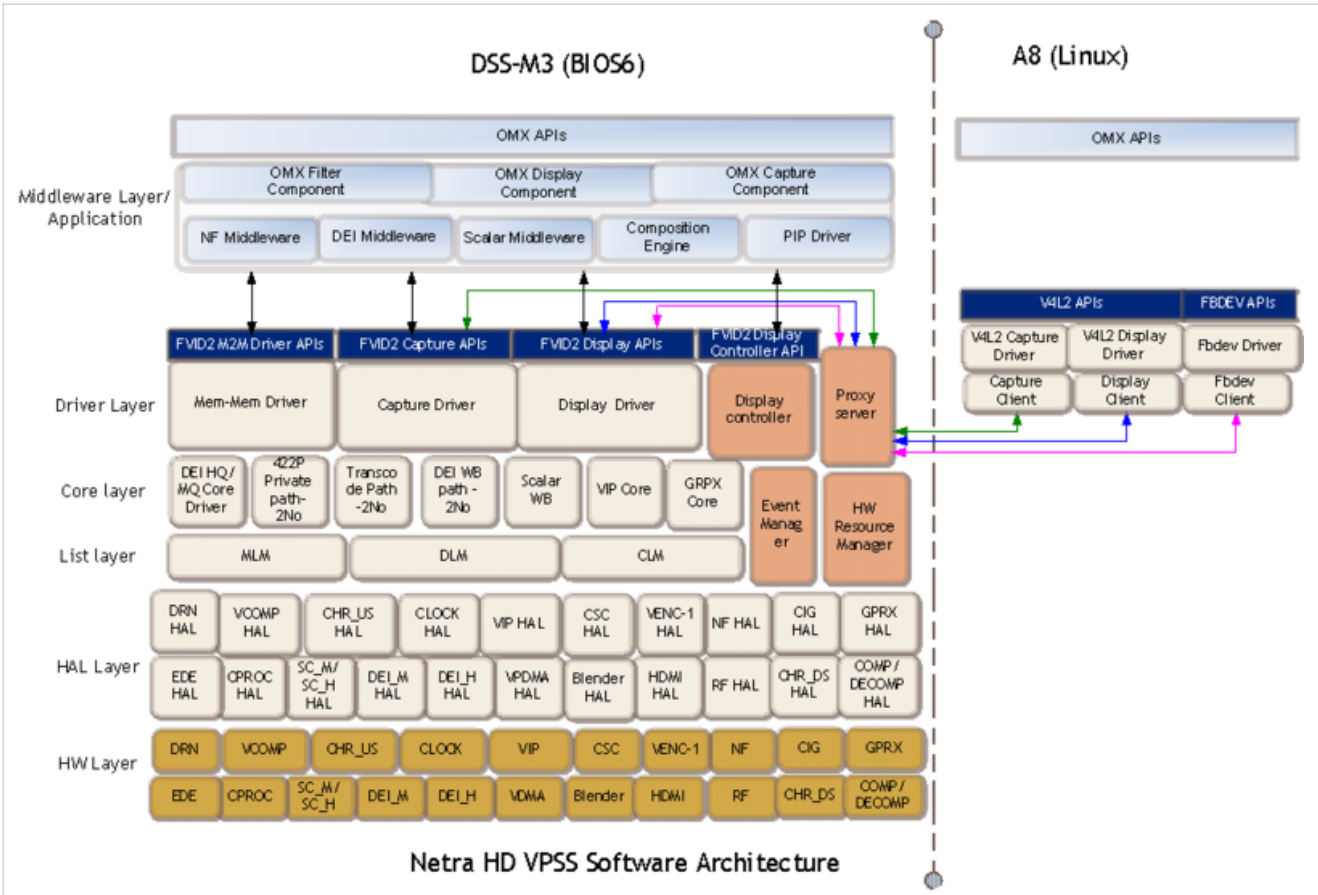
Capture drivers are used for capturing data from external world. Details about each of capture driver features are discussed in the respective section. Only few of important features are mentioned here.

- VIP Capture drivers are non-blocking i.e. asynchronous drivers. Blocking calls are not supported.
- Notification of captured frames is done through callbacks.
- Multi-channel line multiplexed capture - 2CH, 4CH, 8CH - upto D1 (NTSC/PAL) resolution
- Single channel capture upto 1080P (1920x1080) resolution
- Single source (RGB 24-bit or YUV422 8/16-bit), dual output (RGB 24-bit and/or YUV422 and/or YUV420) support
- Multi-instance (VIP0, VIP1), multi-port capture (Port A, Port B), with ability to configure each instance, port independently.
- Capture drivers are supported on:
    - Interfaces : FVID2
    - OS :BIOS6
    - Processor : Ducati M3

HDVPSS Driver Architecture Introduction

It is not necessary to have a full understanding of the HDVPSS driver architecture to use HDVPSS drivers. Only top level driver architecture has been put here. It is sufficient to have understanding of FVID2 interface for development of application or use of HDVPSS drivers. FVID2 interface is explained in details in different section.

HDVPSS Driver architecture follows layered architecture.



DSS Software Architecture

Different layers or components/modules of HDVPSS are:

- **Driver layer**- It is top most layer of HDVPSS driver architecture and FVID2 interface is exposed at this layer. FVID2 interface is used for interaction with application. Apart from exposing FVID2 interface, driver layer is also responsible for queue and de-queuing of request, handling of interaction between cores and list manager layers with driver layer.
- **Core layer**- It is responsible for creation of descriptors. Descriptors are 32 or 16 bytes in length and are created in the memory and provided to VPDMA for final action. Further, descriptors could be understood as command to HDVPSS DMA engine for data transfer or for setting different configuration like height and width of frame.
- **List layer**- These descriptors are required to arranged in specific sequence for different kind of operation or drivers like display, capture and M2M. For example, configuration descriptor for setting frame size should be placed before data descriptor which is responsible for actual data movement. These special arrangement of descriptors for different kind of operations are handled through list manager. As explained above, descriptors could be understood as command to HDVPSS DMA engine for data transfer or for setting different configuration like height and width of frame.
- **HAL layer**- Hardware Abstraction layer – as name suggest – it abstracts multiple IPs of HDVPSS and provides interfaces for upper layer of stacks.
- **HW layer**- It is HDVPSS H/w layer.

- **Event Manager**- There is single interrupt flowing into Ducati M3 for HDVPSS. Event manager parses the interrupt status register to figure out different kind of interrupts and propagates to different modules of HDVPSS stack.
- **Resource Manager**- There are multiple driver possible because of various paths. Different path may use same IP like DEI in DEI_H path in Display and Memory driver. It means only one of driver could active at any point of time. Resource manager handles allocation resources to different drivers.
- **Proxy Server**- HDVPSS stack supports multi core driver architecture. It means that driver could be invoked from different core i.e. using FVID2 interface on M3 hosting BIOS6 or v4l2/fbdev driver on A8 hosting Linux. This has been achieved using IPC communication between M3 and A8.This is internal module of HDVPSS software stack. It listen IPC request from V4l2/fbdev driver. It translates into appropriate FVID2 request and sends to HDVPSS driver on Ducati M3 as if request has originated locally on Ducati M3.It uses Notify again to inform back completion of request.

HDVPSS Driver Co-existence rule

There are several drivers possible out DSS block diagram as discussed earlier. It is very obvious that all drivers can't co-exit mainly because of two reasons.

- **Same IP block required in more drivers** - Same paths can't be used in two drivers at same time. For example, DEI can't be used for M2M de-interlacing and also for online de-interlacing during display at same time. In other words, same IP should not be used by two drivers at same time. There are several such instances where same IP are used in different drivers. This has been managed through **resource manager**. Resource manager checks for availability of resources while opening driver.
- **Number of list –** As discussed earlier, descriptors are used for programming of HDVPSS DMA engine i.e. VPDMA. Descriptors could be understood as command to HDVPSS DMA engine for data transfer or for setting different configuration like height and width of frame. Further to this, these descriptors are required to be arranged in specific sequence for different kind of operation or drivers like display, capture and M2M. These specific arrangements of descriptors in memory should be in contiguous and start address of this buffer holding several descriptors should be given to VPDMA. These contiguous buffers which hold descriptors are called as **list**. HDVPSS could handle such eight **lists** which represents eight different set of descriptors.

These lists are allocated dynamically to different drivers while opening of driver by resource manager. In general,

- One list for each display and there are independent displays are possible. It means that display could take maximum of 3 lists in the case three display are active. This configuration is per TV and does not depend upon how video planes like 422BP or GRPX plane are connected to same TV.
- One list for irrespective of number or kind of captures. In other words, two 1080p capture and 16 channels D1 capture will also use one list.
- Each memory driver requires different list.
- Note that list could be for activating more memory drivers in the case one of TV say SD display is not active.

There is co-existence matrix for drivers which help in showing which driver could co-exist. It could be found at Media:HDVPSS-Coexistence.xls.

T1816X-HDVPSS-HW Overview

TI816X HDVPSS Hardware Introduction

The display sub system includes video display processing modules using the latest TI developed algorithms, flexible compositing and blending engine, full range of external video interfaces in order to deliver a high quality video contents to the end devices. This document covers various aspects of HD-related requirements in addition to SD-related requirements.

Each of the components are explained in detail along with its features in the Netra display subsystem overview documents.

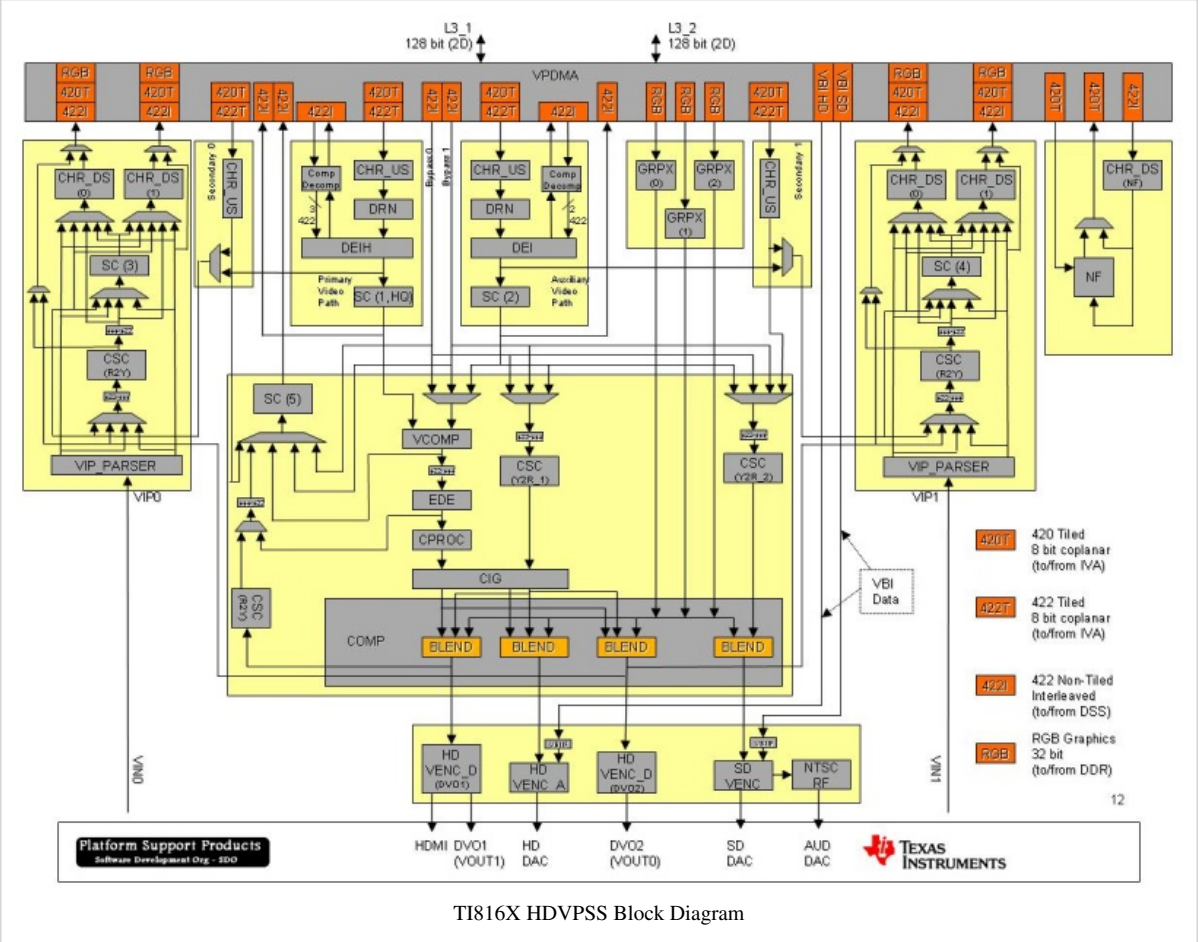
HDVPSS Features

Few top features are mentioned below

- The HDVPSS supports HD (up to 1080p) and SD (NTSC/PAL) outputs simultaneously
- The HDVPSS handles both video and graphics efficiently to create high-quality user interfaces. This includes (but not limited to) deinterlacing, scaling, noise reduction, alpha blending, chroma keying, flicker filtering, and pixel format conversion.
- It supports tiled and raster data formats, scan format conversion, aspect-ratio conversion, and frame size conversion.
- The HDVPSS generate secure video signal with proper content protection mechanisms,i.e., HDCP and Macrovision/CGMS-a for digital and analog outputs, respectively.
- Four independently controlled compositors (HDMI, HD-comp, DVO2, SD) shall be supported.
- Two parallel video processing pipelines (main and aux) for concurrent video stream processing is supported.
- Both the main and auxiliary video pipelines shall include a write-back path to the external memory to support memory to memory scaling of video frames independently from the display output frame timing.
- It supports three graphics plane and include an up/down scaler optimized for graphics application with each graphics path. Multiple regions in a graphics layer are supported to reduce the amount of data transfer from the external memory.
- HDVPSS supports two independently configurable external video input capture ports.Each video input capture port can be operated as one 16/24-bit input channel (with separate Y and Cb/Cr inputs) or two clock independent 8-bit input channels (with interleaved Y/C data input). Embedded sync and external sync modes are supported for all input configurations.
- The video capture port channel shall support de-multiplexing of both pixel-to-pixel and line-to-line multiplexed streams. It could support upto 16 D1 or 32 CIF multiplexed mode capture.It could also support upto 2 channel 1080p60 capture.

HDVPSS Block Diagram

Below figure shows the full blown block diagram of the HDVPSS. Driver uses this diagram to point out the paths used by respective drivers for the different HDVPSS components.



TI816X HDVPSS Block Diagram

DEI_H

The DEI_H (High Quality De-interlacer) is primarily used to convert interlaced video source material to progressive form. This particular module incorporates features such as Temporal Noise Reduction, 4 and 5 field motion detection and very fine edge detection capabilities to produce a very high quality deinterlaced output. In addition, it performs film mode detection and film mode deinterlacing. It can perform deinterlacing on up to 1080i video input source, producing 1080p video output.

DEI_M

The DEI (De-interlacer) is primarily used to convert interlaced video source material to progressive form. This particular module is a reduced feature set of the DEI_H module, in that in does not perform Temporal Noise Reduction and is limited to 4 field motion detection. It performs edge directed interpolation, but utilizes a simpler (and smaller) algorithm compared to the DEI_H module. In addition, it performs film mode detection and film mode deinterlacing. It can perform deinterlacing on up to 1080i video input source, producing 1080p video output.

CHR_US

The CHR_US (Chroma Upsampler) converts YUV420 data format input to YUV422 data format output.

DRN

The DRN (De-Ringing) applies a de-ringing algorithm on input video data to reduce noise.

SC_H

The SC_H (High Quality Scaler) takes the data from the upstream module. The input image is resized to the desired output size. The module sends the output image to the downstream module. It can scale full HD (1080p) and output full HD (1080p) and uses edge-directed vertical scaling to create a high quality result.

SC_M

The SC_M (Scaler) takes the data from the upstream module. The input image is resized to the desired output size. The module sends the output image to the downstream module. It can scale full HD (1080p) and output full HD (1080p).

VCOMP

The VCOMP (Video Compositor) module composites two sources of input video over a background color layer. Both input sources are in 4:2:2 YUV format. The output of the module is also 4:2:2 YUV.

EDE

The EDE (Edge Detail Enhancer) module performs edge detail enhancement on the input video source.

CPROC

Color processing is to provide

- color space conversion,
- dynamic contrast control, and
- color-related processing such as flesh tone detection, memory color enhancement, white point control.

Advanced color processing is performed in the CIE Color Appearance Model 2.0 (CIECAM 2.0).

CIG

The CIG module takes in a single non-constrained video and generates following two outputs:

- The same non-constrained video which may optionally be interlaced
- The same or constrained version of the source video which may optionally interlaced

The first output is sent to the HDMI digital output (via COMP/HD_VENC_D) and the second output is sent to the analog HD component output (via COMP/HD_VENC_A). In addition, the CIG modules takes in a second video input and positions the video in a full display output screen if the input video is a PIP sized.

CSC

The CSC (Color Space Conversion) converts from either YUV444 format to RGB format or RGB format to YUV444 format.

COMP

The COMP (Compositor) blends video from the two video sources with the Graphics sources (GRPX) to form the final video streams going to the three video encoders. COMP has independent compositor/blender, each of them them could upto 5 input layers (2 video and 3 graphics).

GRPX

GRPX is a region-based graphics processor that composes one or more graphics regions to create a display plane input for the video compositor. Regions are rectangular in size. GRPX module could handle multiple rectangular “regions” and composite them into one full screen sized image. GRPX inserts blank pixel data (zero pixel) where region data is unavailable. It supports color formats on Graphics pipeline: RGB565, ARGB1555, RGBA5551, ARGB4444, RGBA4444, ARGB6666,RGBA6666,RGB888, ARGB8888 and RGBA8888, Palette of 1/2/4/8 bits per pixel.

HD_VENC_D_DVO1

The HD_VENC_D_DVO1(High Definition Video Encoder HDMI/DVO1) converts internally processed video to both an HDMI format or DVO format.

HD_VENC_A

The HD_VENC_A (High Definition Video Encoder Analog) converts internally processed video to Component format

HD_VENC_D_DVO2

The HD_VENC_D_DVO2 (Hish Definition Video Encoder DVO2) converts internally processed video to DVO format

SD_VENC

The SD_VENC (Standard Definition Video Encoder) converts internally processed video to composite, S-Video and component format outputs.

NTSC_RF

The NTSC_RF (NTSC R/F Modulator) performs R/F modulation on the output of the SD_VENC

VIP (PARSER)

The VIP Parser (Video Input Port Parser) provides an input for external video sources. Each Video Input Port can receive from 16 CIF streams to 1 1080p60 streams. There are two instance of VIP Parser.

CHR_DS

The CHR_DS (Chroma Downsampler) converts YUV422 data format input to YUV420 data format output.

NF

The NF (Noise Filter) performs a memory to memory spatial/temporal noise filter algorithm on a 422 raster input source and produces a 420 tiled output source. Its primary use mode is part of the Video Input Port processing.

COMP/DECOMP

COMP/DECOMP (Compress/Decompress)are modules that are used to perform compression on DEI private YUV422 private data outbound and decompression on inbound DEI private YUV422 data.

VPDMA

The VPDMA shall be capable of transporting data to and from an external memory location, most often an EMIF, buffering this data and then delivering the data as demanded to Application Modules as programmed.

T1814X-HDVPSS-HW Overview

TI814X/TI8107 HDVPSS Hardware Introduction

The display sub system includes video display processing modules using the latest TI developed algorithms, flexible compositing and blending engine, full range of external video interfaces in order to deliver a high quality video contents to the end devices. This document covers various aspects of HD-related requirements in addition to SD-related requirements.

Each of the components are explained in detail along with its features in the TI814x display subsystem overview documents.

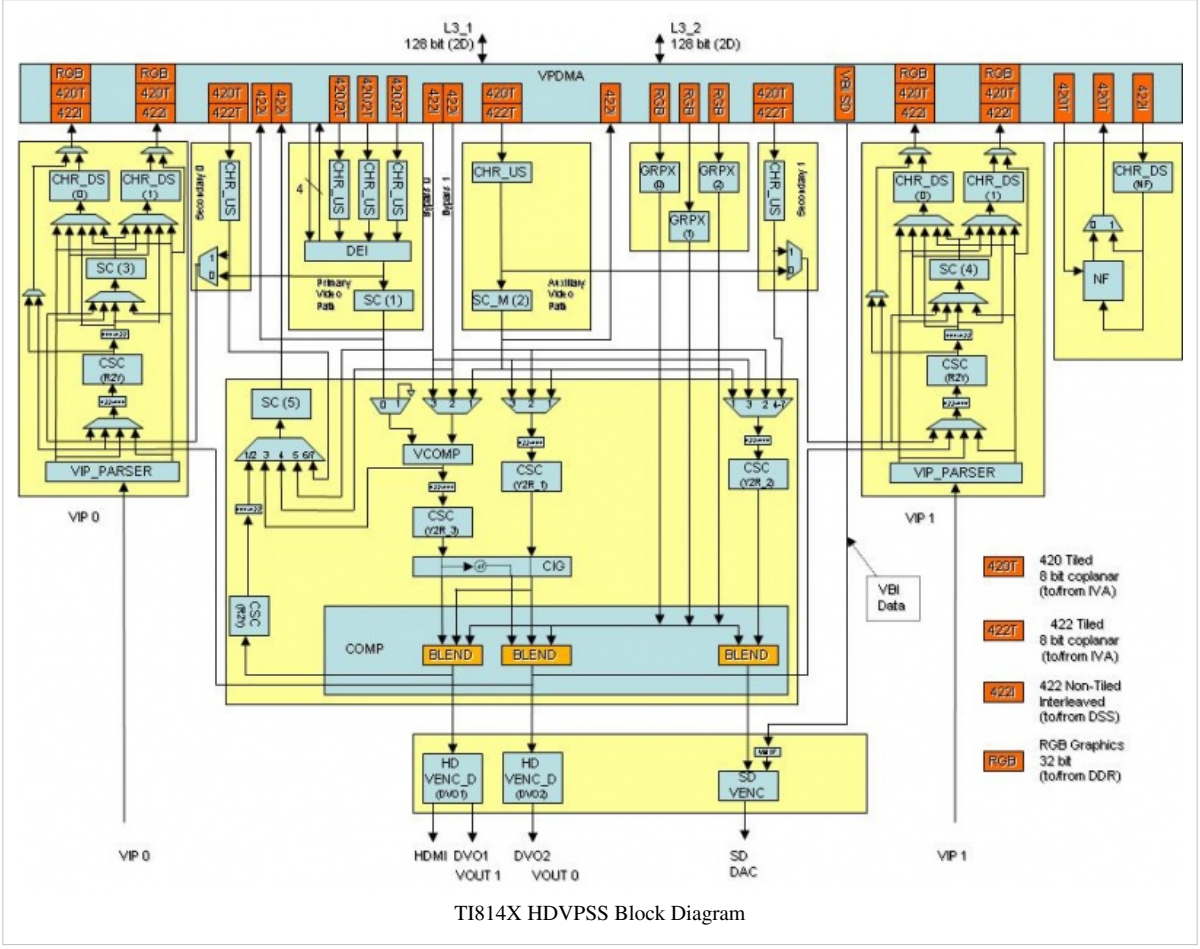
HDVPSS Features

Few top features are mentioned below:

- The HDVPSS supports HD (up to 1080p) and SD (NTSC/PAL) outputs simultaneously
- The HDVPSS handles both video and graphics efficiently to create high-quality user interfaces. This includes (but not limited to) deinterlacing, scaling, noise reduction, alpha blending, chroma keying, flicker filtering, and pixel format conversion.
- It supports tiled and raster data formats, scan format conversion, aspect-ratio conversion, and frame size conversion.
- The HDVPSS generate secure video signal with proper content protection mechanisms,i.e., HDCP and Macrovision/CGMS-a for digital and analog outputs, respectively.
- Three independently controlled compositors (HDMI, DVO2, SD) is supported.
- TI8107 supports four compositors (HDMI, HDCOMP, DVO2, SD) is supported.
- Two parallel video processing pipelines (main and aux) for concurrent video stream processing is supported.
- Both the main and auxiliary video pipelines shall include a write-back path to the external memory to support memory to memory scaling of video frames independently from the display output frame timing.
- It supports three graphics plane and include an up/down scaler optimized for graphics application with each graphics path.
- HDVPSS supports two independently configurable external video input capture ports.Each video input capture port can be operated as one 16/24-bit input channel (with separate Y and Cb/Cr inputs) or two clock independent 8-bit input channels (with interleaved Y/C data input). Embedded sync and external sync modes are supported for all input configurations.
- The video capture port channel shall support de-multiplexing of both pixel-to-pixel and line-to-line multiplexed streams. It could support upto 16 D1 or 32 CIF multiplexed mode capture.It could also support upto 2 channel 1080p60 capture.

TI814x HDVPSS Block Diagram

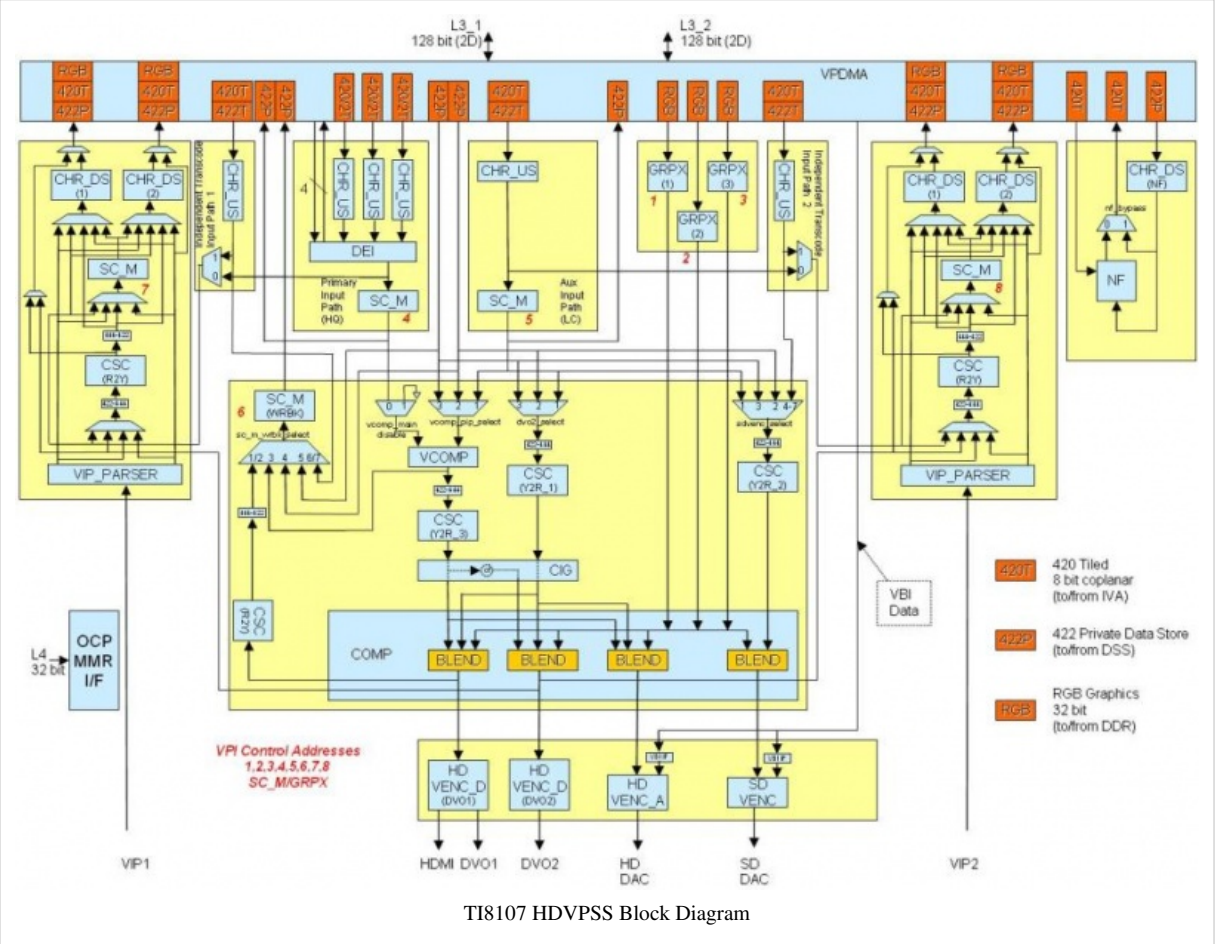
Below figure shows the full blown block diagram of the HDVPSS. Driver uses this diagram to point out the paths used by respective drivers for the different HDVPSS components.



TI814X HDVPSS Block Diagram

TI8107 HDVPSS Block Diagram

Below figure shows the full blown block diagram of the HDVPSS. Driver uses this diagram to point out the paths used by respective drivers for the different HDVPSS components.



TI8107 HDVPSS Block Diagram

DEI

The DEI (De-interlacer) is primarily used to convert interlaced video source material to progressive form. Performs motion adaptive de-interlacing. Supports 4 field motion detection. It performs edge directed interpolation, detects edges in seven direction in 2X7 window. In addition, it performs film mode detection and film mode deinterlacing. It can deinterlace on up to 1080i video input source, producing 1080p video output.

CHR_US

The CHR_US (Chroma Upsampler) converts YUV420 data format input to YUV422 data format output.

SC

The SC (Scaler) takes the data from the upstream module. The input image is resized to the desired output size. The module sends the output image to the downstream module. It can scale full HD (1080p) and output full HD (1080p).

VCOMP

The VCOMP (Video Compositor) module composites two sources of input video over a background color layer. Both input sources are in 4:2:2 YUV format. The output of the module is also 4:2:2 YUV.

CIG

The CIG module takes in a single non-constrained video and generates following two outputs:

- The same non-constrained video which may optionally be interlaced
- The same or constrained version of the source video which may optionally interlaced

The first output is sent to the HDMI digital output (via COMP/HD_VENC_D) and the second output is sent to the analog HD component output (via COMP/HD_VENC_A). In addition, the CIG modules takes in a second video input and positions the video in a full display output screen if the input video is a PIP sized.

CSC

The CSC (Color Space Conversion) converts from either YUV444 format to RGB format or RGB format to YUV444 format.

COMP

The COMP (Compositor) blends video from the two video sources with the Graphics sources (GRPX) to form the final video streams going to the three video encoders. COMP has independent compositor/blender, each of them them could upto 5 input layers (2 video and 3 graphics).

GRPX

GRPX is a region-based graphics processor that composes one or more graphics regions to create a display plane input for the video compositor. Regions are rectangular in size. GRPX module could handle multiple rectangular “regions” and composite them into one full screen sized image. GRPX inserts blank pixel data (zero pixel) where region data is unavailable. It supports color formats on Graphics pipeline: RGB565, ARGB1555, RGBA5551, ARGB4444, RGBA4444, ARGB6666,RGBA6666,RGB888, ARGB8888 and RGBA8888, Palette of 1/2/4/8 bits per pixel.

HD_VENC_D_DVO1

The HD_VENC_D_DVO1(High Definition Video Encoder HDMI/DVO1) converts internally processed video to both an HDMI format or DVO format.

HD_VENC_D_DVO2

The HD_VENC_D_DVO2 (Hish Definition Video Encoder DVO2) converts internally processed video to DVO format

HD_VENC_A

This output is supported only on TI8107. The HD_VENC_A (High Definition Video Encoder Analog) converts internally processed video to Component format. This output runs in synchronous with either HDMI VENC or DVO2 VENC. It is not possible to run this VENC independently.

SD_VENC

The SD_VENC (Standard Definition Video Encoder) converts internally processed video to composite, S-Video and component format outputs. Only Composite output supported on TI8107.

VIP (PARSER)

The VIP Parser (Video Input Port Parser) provides an input for external video sources. Each Video Input Port can receive from 16 CIF streams to 1 1080p60 streams. There are two instance of VIP Parser.

CHR_DS

The CHR_DS (Chroma Downsampler) converts YUV422 data format input to YUV420 data format output.

NF

The NF (Noise Filter) performs a memory to memory spatial/temporal noise filter algorithm on a 422 raster input source and produces a 420 tiled output source. Its primary use mode is part of the Video Input Port processing.

VPDMA

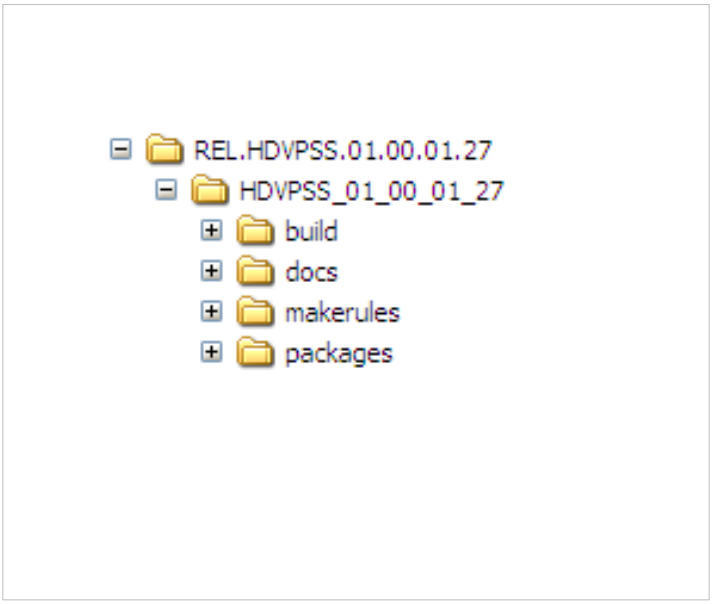
The VPDMA shall be capable of transporting data to and from an external memory location, most often an EMIF, buffering this data and then delivering the data as demanded to Application Modules as programmed.

UserGuideHdvpssFolderOrg

HDVPSS Code / Directory Organization

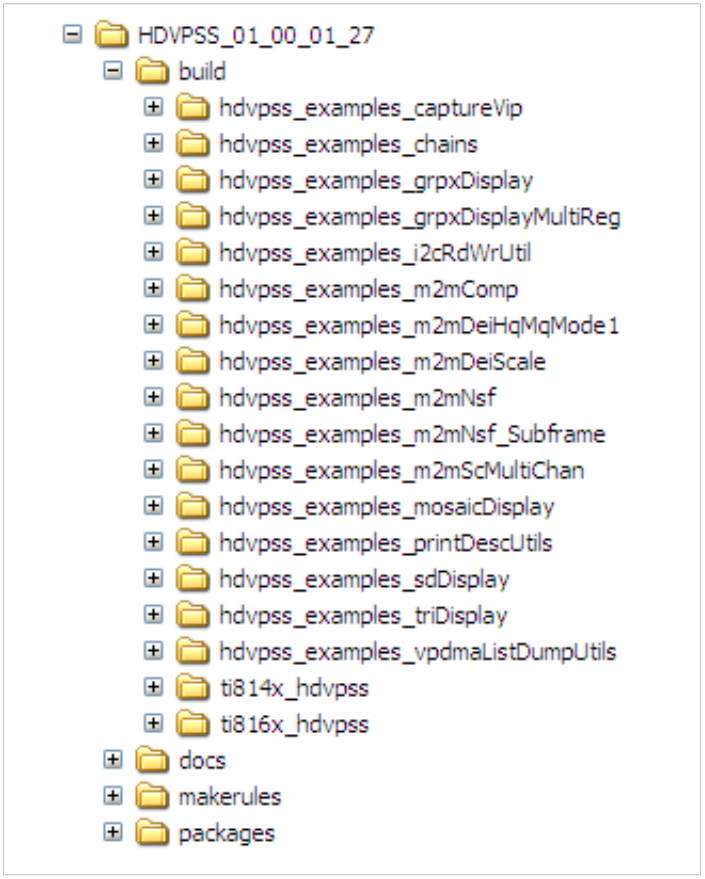
Top Level

On successful installation of HDVPSS source code, in the installed directory following folders would be created. Lets consider 01_00_01.27 versioned release as an example.



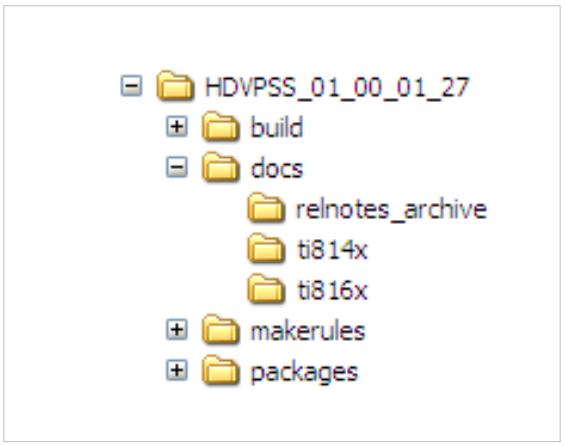
Build

Build directory essentially holds all the generated binaries and libraries. For each sample application that comes with this HDVPSS release, will have separate folder under build directory. Each example in turn will have platform specific folder, which will hold the binary for that platform.



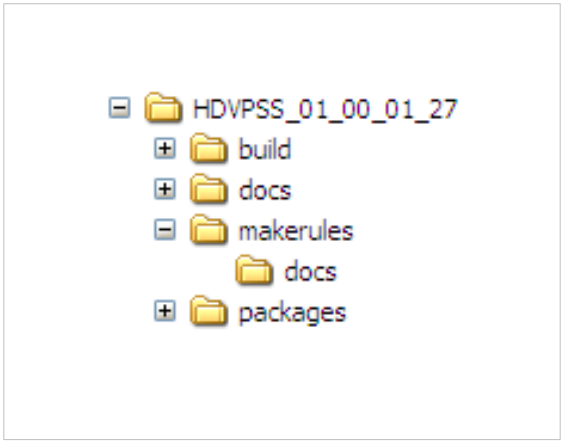
Docs

The docs folder contains release notes, user guide, API guide, gel files and other for all the platforms. The relnotes_archive folder contains the user guides for the previous releases. The platform specific folders (such as ti814x, ti8107, ti816x) contains the gels files (for both A8 and M3 cores), binary to configure on-chip HDMI from A8 and other utilities



Makerules

The folder contains the make files required to compile HDVPSS drivers, sample applications and other utilities. The docs folder under makerules folder contain makerules_spec.doc document that elaborates on the make files used.

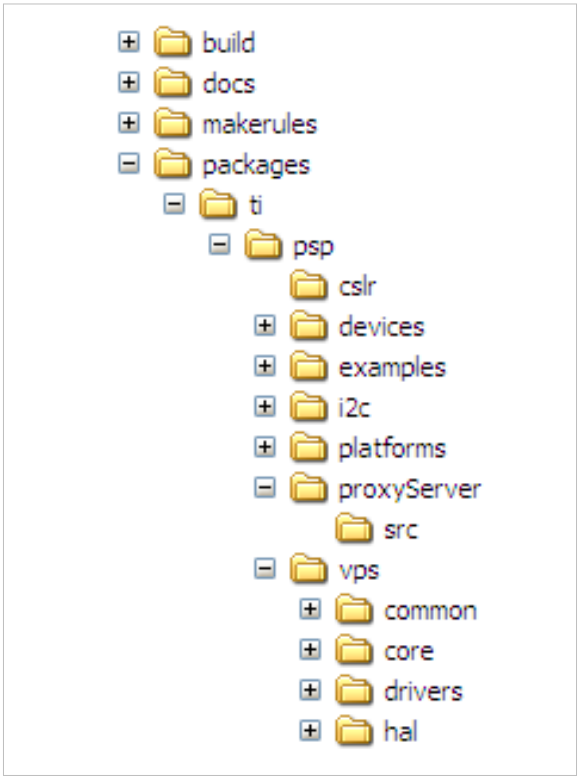


Packages

Detailed in next section - Reducing the indentation by two levels.

Packages

Is the root folder for all HDVPSS Drivers, the following sections expand on contents of sub-folders.



CSLR

Contains defines that would be used to access registers of the individual hardware blocks. The platform specific file, defines the base address of each of the blocks.

Devices

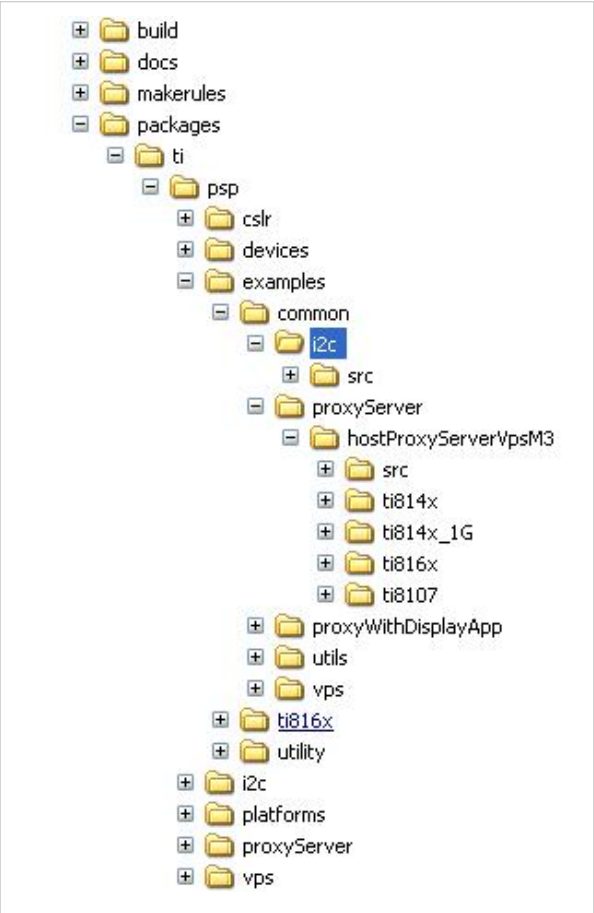
Contains driver / function implementation to control on-board encoders, decoders, other devices such as TVP7002, TVP5158, etc…

Folder devices/src – Contains functions that initialize the supported on-board devices drivers and de-initialize the same. Folder <device name> e.g. TVP7002 – Interface file vpsdrv_tvp7002.h at \packages\ti\psp\devices\tvp7002 - Defines the interface exposed by the TVP7002 decoder driver, typically initialization and de-initialization function called by device initialization. \devices\src\vpsdrv_device.c Encoder / decoder device driver implementation could be found at - \packages\ti\psp\devices\tvp7002\src

Examples

The examples folder is the root folder for all the examples provided in standard HDVPSS release. The examples could be broadly classified into platform specific examples and examples that are common for all supported platforms.

Common



I2C

Implements the command line based I2C application, that could be used to read / write into any of the video on-board devices such as sii9022a, TVP7002, IO Expanders.

Proxy Server

Implements host application for the proxy server. The file ProxyServerHost_main.c implements host at \proxyServer\hostProxyServerVpsM3\src For each of the supported platform, the memory map could be different, the config file proxyServerHost_ti81xx.cfg at \proxyServer\hostProxyServerVpsM3\ti814x\ specifies the memory map for a given platform.

Utils

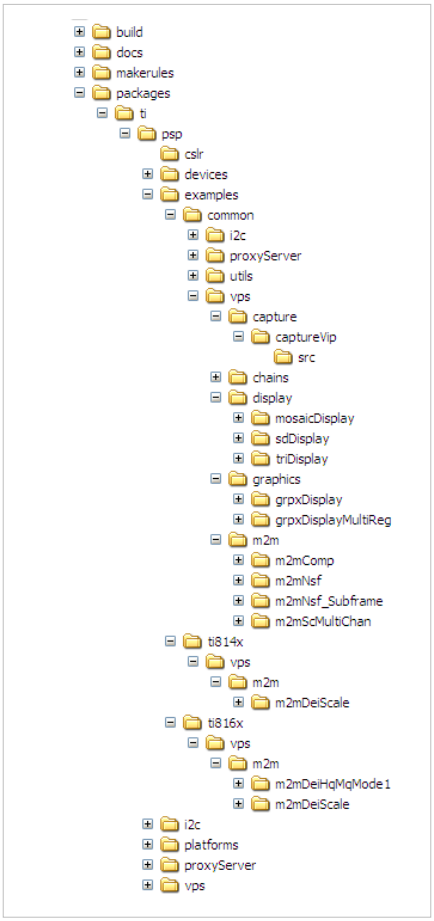
Implements generic utility applications that are supported.

- Utils\printDesc folder – Implements a simple application that could be used to print any VPDMA descriptors. This is typically used as debug aid to identify corrupted / incorrect VPDMA descriptors.
- Utils\vpdmaListDump – Implements a application that could dump the current configuration of the List Manager. This is another debug aid.

VPS

Is the root folder that contains all the driver sample application implementations. The organization of these examples are similar for all examples. The following paragraph expands on one of the example, same could be extended to others

In case a driver has multiple sample applications, there would be multiple folders, each holding a sample application. E.g. Display application, there are 3 different sample application for display, all these sample applications are in separate folders under display folder.



captureVip

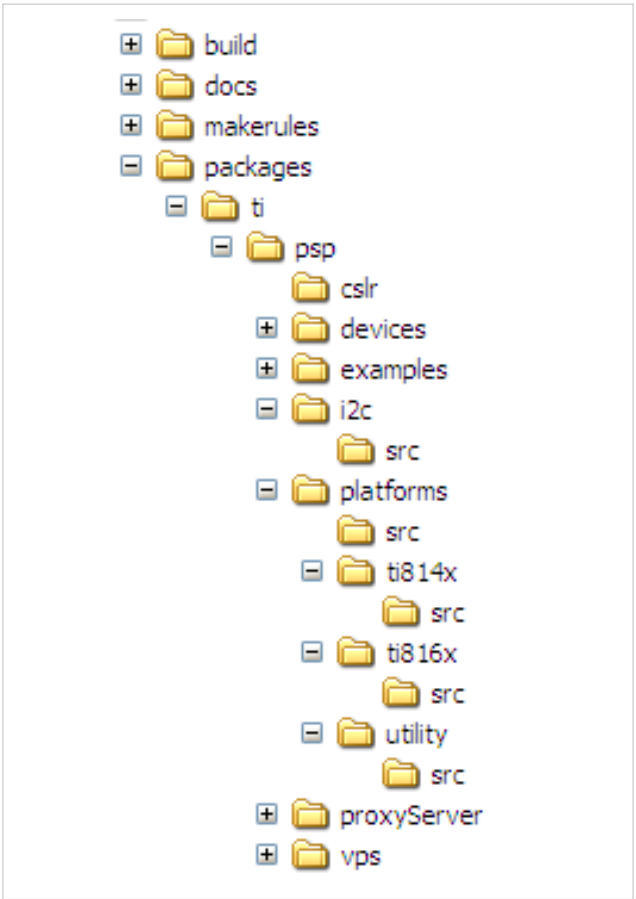
Is the place holder for the make file and captures applications configuration file. The src folder under this holds the source code for the captures sample application. Please refer the user guide of capture for details on the capture sample application.

I2C

The folder is the place holder for I2C driver implementation. The interface exposed by I2C driver is available at \packages\ti\psp\i2c\psp_i2c.h. The source I2C driver is available at \packages\ti\psp\i2c\src\

Platform

The platform specific operations such as determining the platform type, board versions, silicon versions, etc… is implemented by files / functions under this folder. As part of HDVPSS initialization, the platform functionality would be initialized



Utility

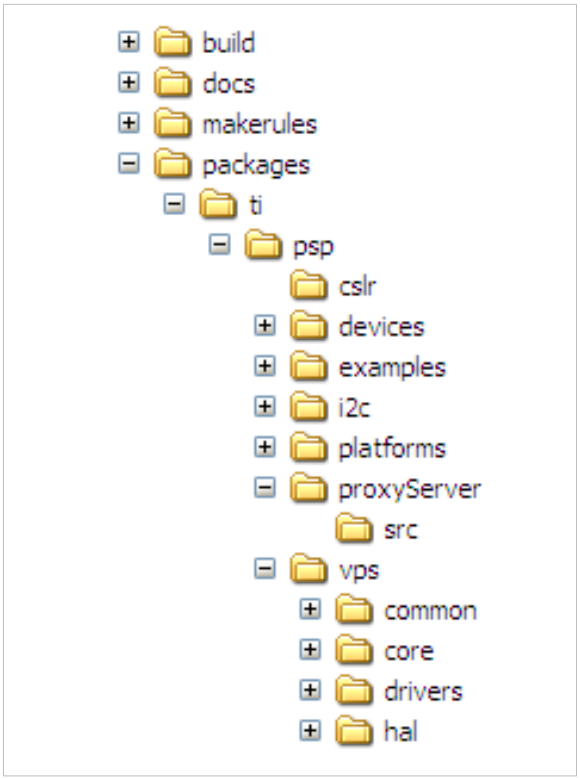
Is the place holder for all generic utility functions (such heap memory, Tiler memory management, etc…)

Proxy Server

Is the place holder for proxy server implementation. The interface exposed by proxy server is at \packages\ti\psp\proxyServer\vps_proxyServer.h, along with make file to build proxy server. The folder \packages\ti\psp\proxyServer\src\ contains the source files for proxy server.

VPS

Is the master folder for all HDVPSS drivers. The interface files required for the entire driver and the FVID 2 interface is available at \packages\ti\psp\vps\. Any application that requires using the services of HDVPSS driver will have to include one or more of the interface files.



Common

The common functionality such as queuing, event management, resource management, event/error logging is implemented in this folder. The files under packages\ti\psp\vps\common\ provides the interfaces that could be used by the drivers. The folder packages\ti\psp\vps\common\src\ is the place holder for the implementation.

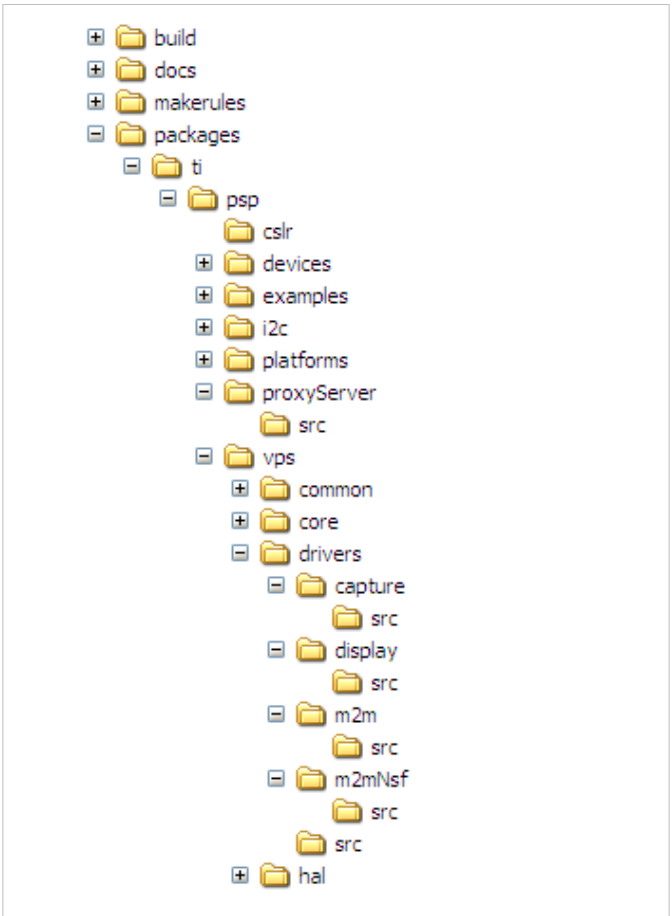
Core

The common functional block / paths that could potentially be used by one or more drivers, is configured / managed by common software entity called “core”. The implementation of this core is at \packages\ti\psp\vps\core\src\ and the interface files of core is at \packages\ti\psp\vps\core

HAL

Is a software abstraction of the underlying hardware, provides function to configure the devices / VPDMA configuration descriptors.

Driver



The HDVPSS drivers are segregated into capture, display and m2m drivers. Each of the sub-folder under driver folder implements one or more HDVPSS drivers. Each of the sub-folder have consistent sub-folder directory structure, as an example, lets consider Capture sub-folder

The application interface exposed by the drivers is available at \packages\ti\psp\vps\ and the driver interface (to FVID2 Manager) is defined at \packages\ti\psp\vps\drivers\capture\

The makefile to build the driver is also contained here.

The source folder “src” under the driver folder is the place holder for the driver implementation. Note that there could multiple drivers, as in case of the m2m driver. The driver instances defined in the application interface at \packages\ti\psp\vps\ is expected to be used to differentiate the drivers.

UserGuideFVID2

FVID2

Introduction

FVID2 are the interface APIs for the video capture, video display and video processing (Memory to Memory drivers)applications on top of BIOS operating system. Provides the interfaces for the streaming operations like queuing of buffers to the hardware and getting is back from the hardware. Also provides the control interface for the devices like video encoders and video decoders which are actually not the data path devices. Gives same look and feel for the video applications across different SoCs.

Following are the features of the FVID2 APIs.

- Platform independent and CPU independent APIs.
- Suitable for multiprocessor communication environment like client-server model.
- Supports blocking as well as non-blocking APIs.
- Supports streaming class of devices like video capture and video display.
- Supports non-steaming class of devices like video encoders and video decoders.
- Supports sliced based operations like sliced based capture and slice based memory to memory drivers.
- Support for the multiple buffers representing a single frame.
- Support for configuring the hardware on per frame basis in synchronous with the frames submitted. AKA Runtime parameters change.
- Interface supports multiple handle and multiple channel operation. Explained in detail in coming sections.
- Support for adding the custom controls specific to the device.

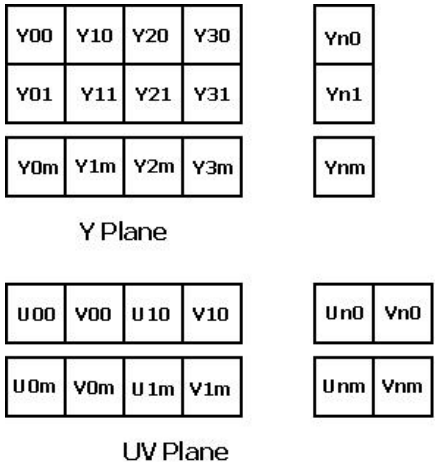
Warning
Underlying drivers catering to FVID2 interfaces may decide to expose the sub-set of features supported by FVID2. Please refer to the individual driver userGuide for the features exposed by drivers.

FVID2 enumerations

FVID2_DataFormat

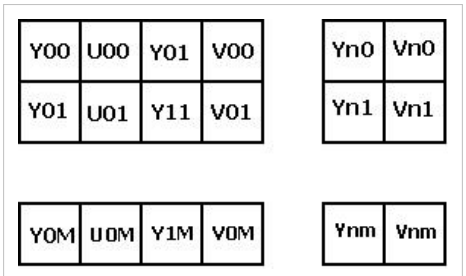
`FVID2_DataFormat` represents the arrangement of the different components forming the pixel. These components can be in YUV color space or the RGB color space or any other color space. Below figure shows the commonly used data formats. FVID2 supports many more data formats. Specific driver may expose subset of the data formats from the mentioned below based on the hardware capability.

YUV420 Semiplanar Format



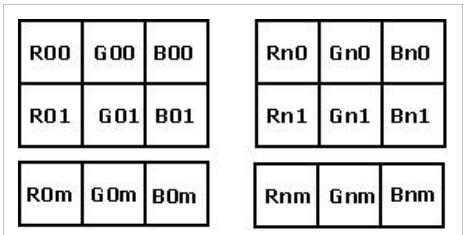
YUV420 DataFormat with Y in one plane and UV in another plane

YUV422 Interleaved Format



YUV422 DataFormat with Y and UV components interleaved

RGB888 Packed Format



RGB888 DataFormat

```
typedef enum
{
    FVID2_DF_YUV422I_UYVY = 0x0000,
        /**< YUV 422 Interleaved format - UYVY. */
    FVID2_DF_YUV422I_YUYV,
        /**< YUV 422 Interleaved format - YUYV. */
    FVID2_DF_YUV422I_YVYU,
        /**< YUV 422 Interleaved format - YVYU. */
    FVID2_DF_YUV422I_VYUY,
        /**< YUV 422 Interleaved format - VYUY. */
    FVID2_DF_YUV422SP_UV,
        /**< YUV 422 Semi-Planar - Y separate, UV interleaved. */
```

```
    FVID2_DF_YUV422SP_VU,
        /**< YUV 422 Semi-Planar - Y separate, VU interleaved. */
    FVID2_DF_YUV422P,
        /**< YUV 422 Planar - Y, U and V separate. */
    FVID2_DF_YUV420SP_UV,
        /**< YUV 420 Semi-Planar - Y separate, UV interleaved. */
    FVID2_DF_YUV420SP_VU,
        /**< YUV 420 Semi-Planar - Y separate, VU interleaved. */
    FVID2_DF_YUV420P,
        /**< YUV 420 Planar - Y, U and V separate. */
    FVID2_DF_YUV444P,
        /**< YUV 444 Planar - Y, U and V separate. */
    FVID2_DF_YUV444I,
        /**< YUV 444 interleaved - YUVYUV... */
    FVID2_DF_RGB16_565 = 0x1000,
        /**< RGB565 16-bit - 5-bits R, 6-bits G, 5-bits B. */
    FVID2_DF_ARGB16_1555,
        /**< ARGB1555 16-bit - 5-bits R, 5-bits G, 5-bits B, 1-bit
Alpha (MSB). */
    FVID2_DF_RGBA16_5551,
        /**< RGBA5551 16-bit - 5-bits R, 5-bits G, 5-bits B, 1-bit
Alpha (LSB). */
    FVID2_DF_ARGB16_4444,
        /**< ARGB4444 16-bit - 4-bits R, 4-bits G, 4-bits B, 4-bit
Alpha (MSB). */
    FVID2_DF_RGBA16_4444,
        /**< RGBA4444 16-bit - 4-bits R, 4-bits G, 4-bits B, 4-bit
Alpha (LSB). */
    FVID2_DF_ARGB24_6666,
        /**< ARGB4444 24-bit - 6-bits R, 6-bits G, 6-bits B, 6-bit
Alpha (MSB). */
    FVID2_DF_RGBA24_6666,
        /**< RGBA4444 24-bit - 6-bits R, 6-bits G, 6-bits B, 6-bit
Alpha (LSB). */
    FVID2_DF_RGB24_888,
        /**< RGB24 24-bit - 8-bits R, 8-bits G, 8-bits B. */
    FVID2_DF_ARGB32_8888,
        /**< ARGB32 32-bit - 8-bits R, 8-bits G, 8-bits B, 8-bit
Alpha (MSB). */
    FVID2_DF_RGBA32_8888,
        /**< RGBA32 32-bit - 8-bits R, 8-bits G, 8-bits B, 8-bit
Alpha (LSB). */
    FVID2_DF_BITMAP8 = 0x2000,
        /**< BITMAP 8bpp. */
    FVID2_DF_BITMAP4_LOWER,
        /**< BITMAP 4bpp lower address in CLUT. */
    FVID2_DF_BITMAP4_UPPER,
```

```
        /**< BITMAP 4bpp upper address in CLUT. */
    FVID2_DF_BITMAP2_OFFSET0,
        /**< BITMAP 2bpp offset 0 in CLUT. */
    FVID2_DF_BITMAP2_OFFSET1,
        /**< BITMAP 2bpp offset 1 in CLUT. */
    FVID2_DF_BITMAP2_OFFSET2,
        /**< BITMAP 2bpp offset 2 in CLUT. */
    FVID2_DF_BITMAP2_OFFSET3,
        /**< BITMAP 2bpp offset 3 in CLUT. */
    FVID2_DF_BITMAP1_OFFSET0,
        /**< BITMAP 1bpp offset 0 in CLUT. */
    FVID2_DF_BITMAP1_OFFSET1,
        /**< BITMAP 1bpp offset 1 in CLUT. */
    FVID2_DF_BITMAP1_OFFSET2,
        /**< BITMAP 1bpp offset 2 in CLUT. */
    FVID2_DF_BITMAP1_OFFSET3,
        /**< BITMAP 1bpp offset 3 in CLUT. */
    FVID2_DF_BITMAP1_OFFSET4,
        /**< BITMAP 1bpp offset 4 in CLUT. */
    FVID2_DF_BITMAP1_OFFSET5,
        /**< BITMAP 1bpp offset 5 in CLUT. */
    FVID2_DF_BITMAP1_OFFSET6,
        /**< BITMAP 1bpp offset 6 in CLUT. */
    FVID2_DF_BITMAP1_OFFSET7,
        /**< BITMAP 1bpp offset 7 in CLUT. */
    FVID2_DF_BAYER_RAW = 0x3000,
        /**< Bayer pattern. */
    FVID2_DF_RAW_VBI,
        /**< Raw VBI data. */
    FVID2_DF_RAW,
        /**< Raw data - Format not interpreted. */
    FVID2_DF_MISC,
        /**< For future purpose. */
    FVID2_DF_INVALID,
        /**< Invalid data format. Could be used to initialize
variables. */
    FVID2_DF_MAX
        /**< Should be the last value of this enumeration.
         Will be used by driver for validating the input parameters. */
} FVID2_DataFormat;
```

FVID2 ScanFormat

Strucutre represents the scanning format.

```
typedef enum
{
    FVID2_SF_INTERLACED = 0,
        /**< Interlaced mode. */
    FVID2_SF_PROGRESSIVE,
        /**< Progressive mode. */
    FVID2_SF_MAX
        /**< Should be the last value of this enumeration.
         Will be used by driver for validating the input parameters. */
} FVID2_ScanFormat;
```

FVID2 Field ID

Represents field ID of the buffer. For interlaced buffers field ID could be 0 or 1 depending upon the even and odd field buffer contains. For progressive displays field ID is same for all the frames.

```
typedef enum
{
    FVID2_FID_TOP = 0,
        /**< Top field. */
    FVID2_FID_BOTTOM,
        /**< Bottom field. */
    FVID2_FID_FRAME,
        /**< Frame mode - Contains both the fields or a progressive
frame. */
    FVID2_FID_MAX
        /**< Should be the last value of this enumeration.
         Will be used by driver for validating the input parameters. */
} FVID2_Fid;
```

Bits per Pixel

Represents bits per pixel for buffer. For example for YUV422 interlaced format bit per pixel will be 16 and for YUV444 it will be 24 and YUV420 it will be 12.

```
typedef enum
{
    FVID2_BPP_BITS1 = 0,
    /**< 1 Bits per Pixel. */
    FVID2_BPP_BITS2,
    /**< 2 Bits per Pixel. */
    FVID2_BPP_BITS4,
    /**< 4 Bits per Pixel. */
    FVID2_BPP_BITS8,
    /**< 8 Bits per Pixel. */
    FVID2_BPP_BITS12,
    /**< 12 Bits per Pixel - used for YUV420 format. */
```

```
    FVID2_BPP_BITS16,
    /**< 16 Bits per Pixel. */
    FVID2_BPP_BITS24,
    /**< 24 Bits per Pixel. */
    FVID2_BPP_BITS32,
    /**< 32 Bits per Pixel. */
    FVID2_BPP_MAX
    /**< Should be the last value of this enumeration.
         Will be used by driver for validating the input parameters. */
} FVID2_BitsPerPixel;
```

FVID2 Structures

FVID2 CallBack Parameters

FVID2 supports the driver call back. Driver call the application on specific events like, completion of buffer capture, displayed or process. Or in case of error where application needs to take some action. Following is the structure defined by the FVID2 API for the application to pass the callback functions to be invoked by the driver.

```
typedef struct
{
    FVID2_CbFxn         cbFxn;
        /**< Application callback function used by the driver to
intimate any
             operation has completed or not. This is an optional
parameter
             in case application decides to use polling method and so
could be
             set to NULL. */
    FVID2_ErrCbFxn      errCbFxn;
        /**< Application error callback function used by the driver
to intimate
             any error occurs at the time of streaming. This is an
optional
             parameter in case application decides not to get any error
 callback
             and so could be set to NULL. */
    Ptr                 errList;
        /**< Pointer to a valid framelist (FVID2_FrameList) in case
of capture
             and display drivers or a pointer to a valid processlist
             (FVID2_ProcessList) in case of M2M drivers where the
driver copies
             the aborted/error packet. The memory of this list should
be
             allocated by the application and provided to the driver at
 the time
             of driver creation. When the application gets this
```

```
callback, it has
             to empty this list and taken necessary action like freeing
 up memories
             etc. The driver will then reuse the same list for future
error
             callback.
             This could be NULL if errCbFxn is NULL. Otherwise this
should be
             non-NULL. */
    Ptr                 appData;
        /**< Application specific data which is returned in the
callback function
             as it is. This could be set to NULL if not used. */
    Ptr                 reserved;
        /**< For future use. Not used currently. Set this to NULL.
*/
} FVID2_CbParams;
```

FVID2 Format

Defines the format capabilities of the buffer like dataformat, scanFormat, width, height etc.

```
typedef struct
{
    UInt32              channelNum;
    /**< Channel Number to which this format belongs to.
         This is used in case of multiple buffers queuing/deqeuing
using a
         single call. This is not applicable for all the drivers. When
not
         used set it to zero. */

    UInt32              width;
    /**< Width in pixels. */

    UInt32              height;
    /**< Number of lines per frame. For interlaced mode, this should
 be set to
         the frame size and not the field size. */

    UInt32              pitch[FVID2_MAX_PLANES];
    /**< Pitch in bytes for each of the sub-buffers. This represents
 the
         difference between two consecutive line address.
         This is irrespective of whether the video is interlaced or
         progressive and whether the fields are merged or separated for
         interlaced video. */
    UInt32              fieldMerged[FVID2_MAX_PLANES];
    /**< Whether both the fields have to be merged - line
```

```
interleaved or not.
         Used only for interlaced format. The effective pitch is
calculated
         based on this information along with pitch parameter. If
fields are
         merged, effective pitch = pitch * 2 else effective pitch =
pitch. */

    UInt32              dataFormat;
    /**< Frame data Format. For valid values see #FVID2_DataFormat.
*/
    UInt32              scanFormat;
    /**< Scan Format. For valid values see #FVID2_ScanFormat. */
    UInt32              bpp;
    /**< Number of bits per pixel. For valid values see
#FVID2_BitsPerPixel. */
    Ptr                 reserved;
    /**< For future use. Not used currently. Set this to NULL. */
} FVID2_Format;
```

FVID2 Slice information

Represents the slice information. Used in sliced bases processing like slice based capture and sliced based memory to memory driver

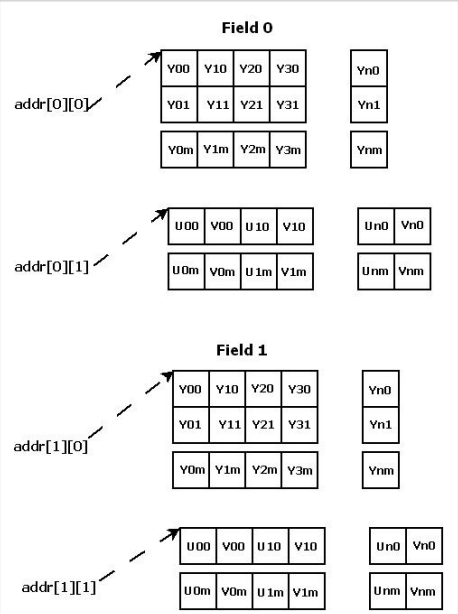
```
typedef struct
{
    UInt32              sliceNum;
        /**< Current slice Number in this frame,
             range is from 0 to (NoOfSlicesInFrame-1)  */
    UInt32              numSlcInLines;
        /**< Number of lines available in the frame at the end of
this slice. */
    UInt32              numSlcOutLines;
        /**< Number of lines generated in output buffer after
processing
             current slice */
} FVID2_SliceInfo;
```

FVID2 Frame

Represents the attribute of one buffer in frame. Attributes like address of each planes and each fields. YUV420 semi-planar buffer with interlaced scan format will have two planes one each for Y data and UV data and odd and even fields. Below figure shows the manipulation of the `addr` field of the FVID2_Frame structure for different data formats and scan formats.

YUV420 Semiplanar

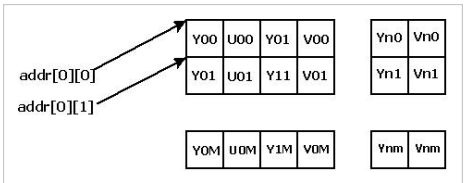
Below figure shows the `addr` field of the FVID2_Frame structure for the YUV420 semi planar data in which the image is interlaced and fields are separate in two different buffers.



`addr` field of FVID2_Frame structure

YUV422 Interleaved

Below figure shows the `addr` field of the FVID2_Frame structure for the YUV422 interleaved data in which the image is interlaced and fields are merged in single buffer. For the progressive image only the `addr[0][0]` needs to be initialized



`addr` field of FVID2_Frame structure

Below figure shows the FVID2 frame structure in detail

```
typedef struct
{
    Ptr                 addr[FVID2_MAX_FIELDS][FVID2_MAX_PLANES];
        /**< FVID2 buffer pointers for supporting multiple addresses
 like
             y, u, v etc for a given frame. The interpretation of these
 pointers
             depend on the format configured.
             The first dimension represents the field and the second
```

```
dimension
             represents the plane. Not all pointers are valid for a
given format.

             Representation of YUV422 Planar Buffer:
             Field 0 Y -> addr[0][0], Field 1 Y -> addr[1][0]
             Field 0 U -> addr[0][1], Field 1 U -> addr[1][1]
             Field 0 V -> addr[0][2], Field 1 V -> addr[1][2]

             Representation of YUV422 Interleaved Buffer:
             Field 0 YUV -> addr[0][0], Field 1 YUV -> addr[1][0]
             Other pointers are not valid.

             Representation of RGB888 Buffer (Assuming RGB is always
progressive):
             RGB -> addr[0][0]
             Other pointers are not valid.

             Instead of using numerical for accessing the buffers, the
application
             can use the macros defined for each buffer formats like
             FVID2_YUV_INT_ADDR_IDX, FVID2_RGB_ADDR_IDX, FVID2_FID_TOP
etc.

             [IN] for queue operation.
             [OUT] for dequeue operation. */
    UInt32              fid;
        /**< Indicates whether this frame belong to top or bottom
field.
             For valid values see #FVID2_Fid.
             [IN] for queue operation.
             [OUT] for dequeue operation. */
    UInt32              channelNum;
        /**< Channel number to which this FVID2 frame belongs to.
             This is used in case of multiple buffers queuing/deqeuing
using a
             single call.
             If only one channel is supported, then this should be set
to zero.
             [IN] for queue operation.
             [OUT] for dequeue operation. */

    UInt32              timeStamp;
        /**< Time Stamp for captured or displayed frame.
             [OUT] for dequeue operation. Not valid for queue
operation. */
    Ptr                 appData;
```

```
        /**< Additional application parameter per frame. This is not
 modified by
             driver. */
    Ptr                 perFrameCfg;
        /**< Per frame configuration parameters like scaling ratio,
positioning,
             cropping etc...
             This could be set to NULL if not used.
             [IN] for queue operation. Dequeue returns the same pointer
 back to
             the application. */
    Ptr                 blankData;
        /**< Blanking data.
             This could be set to NULL if not used.
             [IN] for queue operation.
             [OUT] for dequeue operation. */
    Ptr                 drvData;
        /**< Used by driver. Application should not modify this. */
    FVID2_SliceInfo    *sliceInfo;
        /**< Used for Slice level processing information exchange
between
             application and driver.
             This could be set to NULL if slice level processing is not
 used. */
    Ptr                 reserved;
        /**< For future use. Not used currently. Set this to NULL.
*/
} FVID2_Frame;
```

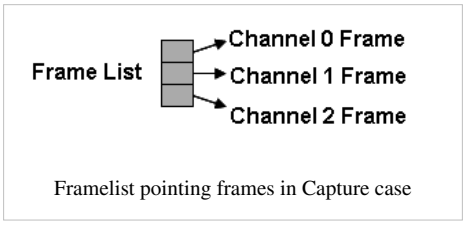
FVID2 FrameList

Framelist represents N frames. For display N frames represent buffer address of each window in a multi-window mode. For capture it represents different channel buffers for the multiplexed channels. Currently FVID2_Framelist can handle maximum of `FVID2_MAX_FVID_FRAME_PTR` frame pointers.

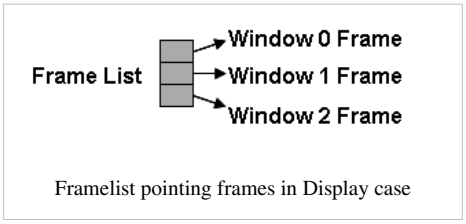
```
typedef struct
{
    FVID2_Frame        *frames[FVID2_MAX_FVID_FRAME_PTR];
        /**< An array of FVID2 frame pointers.
             [IN] The content of the pointer array i.e FVID2_Frame
pointer is input
             for queue operation
             [OUT] Output for dequeue operation. */
    UInt32              numFrames;
        /**< Number of frames - Size of the array containing FVID2
pointers.
             [IN] for queue operation.
             [OUT] for dequeue operation. */
    Ptr                 perListCfg;
```

```
        /**< Per list configuration parameters like scaling ratio,
positioning,
             cropping etc for all the frames together.
             This could be set to NULL if not used. In this case, the
driver will
             use the previous configuration.
             [IN] for queue operation. Dequeue returns the same pointer
 back to
             the application. */
    Ptr                 drvData;
        /**< Used by driver. Application should not modify this. */
    Ptr                 reserved;
        /**< For future use. Not used currently. Set this to NULL.
*/
} FVID2_FrameList;
```

Below figure shows the framelist containing FVID2_Frame in case of display and capture drivers.



Framelist pointing frames in Capture case



Framelist pointing frames in Display case

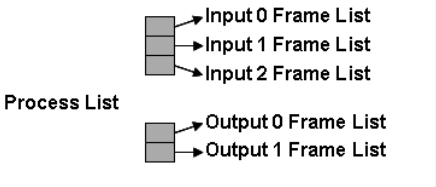
FVID2 ProcessList

FVID2 process list containing frame list used to exchange multiple input/output buffers in M2M (memory to memory) operation. Each of the frame list in turn have multiple frames/request.

```
typedef struct
{
    FVID2_FrameList    *inFrameList[FVID2_MAX_IN_OUT_PROCESS_LISTS];
    /**< Pointer to an array of FVID2 frame list pointers for input
nodes.
         [IN] for both queue and dequeue operation.
         The content of the pointer array i.e FVID2_FrameList pointer
is
         input for queue operation and is output for dequeue operation.
 */
    FVID2_FrameList    *outFrameList[FVID2_MAX_IN_OUT_PROCESS_LISTS];
    /**< Pointer to an array of FVID2 frame list pointers for output
 nodes.
```

```
         [IN] for both queue and dequeue operation.
         The content of the pointer array i.e FVID2_FrameList pointer
is
         input for queue operation and is output for dequeue operation.
 */
    UInt32              numInLists;
    /**< Number of input frame list valid in inFrameList.
         [IN] for queue operation.
         [OUT] for dequeue operation. */
    UInt32              numOutLists;
    /**< Number of output frame list valid in outFrameList.
         [IN] for queue operation.
         [OUT] for dequeue operation. */
    Ptr                 drvData;
    /**< Used by driver. Application should not modify this. */
    Ptr                 reserved;
    /**< For future use. Not used currently. Set this to NULL. */
} FVID2_ProcessList;
```

Below figure shows the processlist containing FVID2_FrameList which in turn will point to FVID2_Frame



ProcessList pointing FVID2_FrameList

FVID2 APIs

FVID2 Init

This API should be called before calling any of the FVID2 APIs. This API initializes the underlying hardware/software sub-system built on top of FVID2 APIs. This should be called once during the system initialization time in the task context. This function should not be called from the ISR context.

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

FVID2 DeInit

This function should be called during the system de-Initialization. De-Initializes the hardware/software sub-system built on top of FVID2 APIs. This should be called only once from the task context.

```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

FVID2 Create

This API is used to open the FVID2 driver. drvId and InstanceId pair represents the hardware on which driver operates. It initializes the hardware supported by the driver and configures it according to the parameters provided by open. Some of the FVID2 driver supports multiple creates/open on the same drvId and instanceId. Requests from the different handles of the multiple opens is serialize by the driver and is operated upon the same hardware one by one.

```
FVID2_Handle FVID2_create(UInt32 drvId,
                          UInt32 instanceId,
                          Ptr createArgs,
                          Ptr createStatusArgs,
                          const FVID2_CbParams *cbParams);
```

drvId - [IN] Used to find a matching ID in the device driver table
instanceId - [IN] Instance ID of the driver to open and is used to differentiate multiple instance support on a single driver.
createArgs - [IN] Pointer to the create argument structure. The type of the structure is defined by the specific driver. This parameter could be NULL depending on whether the actual driver forces it or not.
createStatusArgs - [OUT] Pointer to status argument structure where the driver returns any status information. The type of the structure is defined by the specific driver. This parameter could be NULL depending on whether the actual driver forces it or not.
cbParams - Application callback parameters FVID2_CbParams. This parameter could be NULL depending on whether the actual driver forces it or not.
return - Returns a non-NULL FVID2_Handle object on success else returns NULL on error.

FVID2 Set Format

Sets the format information for the already opened driver for a given channel. This function should be called from the task context.

```
Int32 FVID2_setFormat(FVID2_Handle handle, FVID2_Format *fmt)
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
fmt - [IN] Pointer to the FVID2 Create structure.
return - FVID2_SOK on success, else appropriate FVID2 Error Code on failure

FVID2 Get Format

Returns the format already set for the opened driver for a given channel. This function should be called from the task context.

```
Int32 FVID2_setFormat(FVID2_Handle handle, FVID2_Format *fmt)
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
fmt - [OUT] Pointer to the FVID2 Create structure.
return - FVID2_SOK on success, else appropriate FVID2 Error Code on failure.

FVID2 Control

Driver exposes the custom control commands specific to the driver and hardware througth this interface. All the FVID2 control commands are blocking. These control commands should be called from the task context unless specified otherwise by the specific drivers. Example of the control commands exposed by different drivers are creation/selection of the different multi window layout in case of display driver, programming of coefficients in case of memory drivers involving scalars.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
cmd - [IN] IOCTL command. The type of command supported is defined by the specific driver.
cmdArgs - [IN] Pointer to the command argument structure. The type of the structure is defined by the specific driver for each of the supported IOCTL. This parameter could be NULL depending on whether the actual driver forces it or not.
cmdStatusArgs - [OUT]Pointer to status argument structure where the driver returns any status information. The type of the structure is defined by the specific driver for each of the supported IOCTL. This parameter could be NULL depending on whether the actual driver forces it or not.
return - FVID2_SOK on success, else appropriate FVID2 Error Code on failure.

FVID2 Start

An application calls FVID2 start to request the video device driver to start the video display or capture operation. Most of the control commands and start FVID2 commands like FVID2_setFormat,FVID2_getFormat cannot be called unless specified otherwise by driver. This function should be called from the task context.

```
Int32 FVID2_start(FVID2_Handle handle, Ptr cmdArgs)
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
cmdArgs - [IN] Pointer to the start argument structure. The type of the structure is defined by the specific driver. This parameter could be NULL depending on whether the actual driver forces it or not.
return - FVID2_SOK on success, else appropriate FVID2 Error Code on failure.

FVID2 Stop

An application calls the FVID2 stop to request the video device driver to stop the video display or capture operation. FVID2 Stop may be called by application to change the setting of the driver like format, encoder/decoder mode etc. After doing the required operation driver can be start again.

Warning: If driver settings are called after FVID2_Stop, then remaining buffers in the queue should be de-queued before starting the driver again.

```
Int32 FVID2_stop(FVID2_Handle handle, Ptr cmdArgs)
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
cmdArgs - [IN] Pointer to the start argument structure. The type of the structure is defined by the specific driver. This parameter could be NULL depending on whether the actual driver forces it or not.
return - FVID2_SOK on success, else appropriate FVID2 Error Code on failure.

FVID2 Queue

This is used to submit a video buffer to the video device driver. This is used in capture/display drivers. This function should be called from task context unless driver specifies that it can be called from the interrupt context as well. This is a non blocking API unless the specific driver specifies otherwise.

```
Int32 FVID2_queue(FVID2_Handle handle,
                  FVID2_FrameList *frameList,
                  UInt32 streamId);
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
frameList - [IN] Pointer to the FVID2 FrameList structure containing the information about the FVID2 frames that has to be queued in the driver.
streamId- Stream ID to which the frames should be queued. This is used in drivers where they could support multiple streams for the same handle. Otherwise this should be set to zero.
return - FVID2_SOK on success, else appropriate FVID2 Error Code on failure.

FVID2 De-Queue

An application calls FVID2_dequeue to request the video device driver to give ownership of a video buffer. This is used in the capture and display driver. This is a non-blocking API if timeout is FVID2_TIMEOUT_NONE and could be called by task context as well as interrupt context unless specific driver mentions otherwise. This is blocking API if timeout is FVID2_TIMEOUT_FOREVER if supported by specific driver implementation.

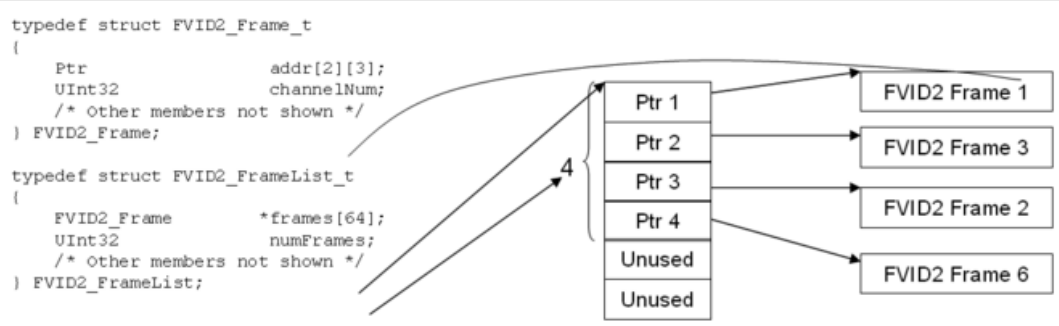
```
Int32 FVID2_dequeue(FVID2_Handle handle,
                    FVID2_FrameList *frameList,
                    UInt32 streamId,
                    UInt32 timeout);
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
frameList - [OUT] Pointer to the FVID2 FrameList structure where the de-queued frame pointers will be stored
streamId - [IN] Stream ID from where frames should be dequeued. This is used in drivers where it could support multiple streams for the same handle. Otherwise this should be set to zero.
timeout - [IN] FVID2 timeout in units of OS ticks. This will determine the timeout value till the driver will block for a free or completed buffer is available. For non-blocking drivers this parameter might be ignored. **return** - FVID2_SOK on success, else appropriate FVID2 Error Code on failure.

FVID2 Queue and De-Queue

Single Queue and Single De-Queue

Single queue and corresponding single de-queue of the framelist is used in the display driver. Where the single framelist can contain the single buffer for the whole frame or can contain multiple buffers in case of multiple window configuration. Below figure shows how FVID2 FrameList and FVID2 Frames are initialized in case of multiple window configuration.

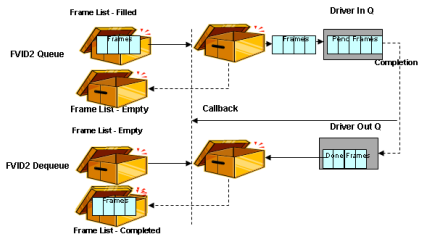


FrameList and Frame Initlialization

As shown in above figure

- One FVID2_Frame is pointing to one buffer each.
- All the FVID2 frames to be display as a part of single video frame is pointed by the FVID2 Frame pointers inside FVID2 FrameList.

Below figure shows how the FVID2 Frame pointers inside the FVID2_Frame list are exchanged between the driver and the application in the FVID2 Queue and FVID2 De-Queue calls.



FVID2_Queue and FVID2_DeQueue sequence in case of Display

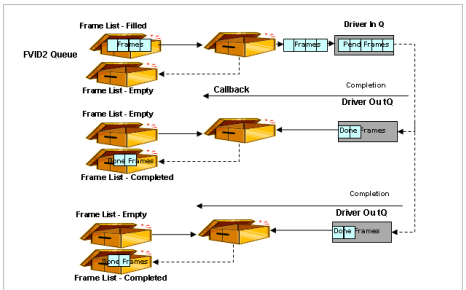
As show in above figure.

- FVID2 FrameList contains 4 FVID2 Frames.
- Its submitted through single FVID2 Queue and will be displayed as single video frame.
- Driver copies all the content inside the FVID2 FrameList into the driver's FVID2 FrameList and application can't touch it till driver returns it back. Now the application FVID2 FrameList is free to load new FVID2 Frames.
- Driver gives the callback to the application on successfully displaying the video frames inside FVID2 FrameList
- Application calls the FVID2 De-Queue with the empty FVID2 FrameList. Driver copied back all the FVID2 Frames back.
- In display case application always queues all the frames required to display one video frame and driver gives it back once it completes displaying that video frame.
- Hence always single FVID2 Queue call results in single FVID2 De-Queue call.

Single Queue and Multiple De-Queue

This is used in case of multiple channel case. While priming of the buffers before the capture starts application submits buffers for all the channels using a single FVID2 Queue call. Since the capture is multiplexed input frames from the different sources could complete at different time for each input and application wants to process buffer as soon as its captured. This concept allows buffers to be de-queued as they are complete without waiting for other channels to be completed. This results in single queue where buffers for all the channels are queued in single called and de-queued as the channels are completed capturing.

Below figure shows the single queue and multiple de-queue used in capture driver.



FVID2_Queue and FVID2_DeQueue sequence in case of Capture

As show in above figure

- FVID2 FrameList contains 4 FVID2 Frames one for each channel in case of 4 channels multiplexed capture.
- Capture driver gives callback to the application with two frames completed capturing.
- Application calls FVID2 De-Queue with empty FVID2 FrameList.
- Capture driver returns pointers to both completed FVID2 Frames
- Again capture driver gives callback to application with the rest of the two frames captured.
- Application calls FVID2 De-Queue with empty FVID2 FrameList.
- Again capture driver returns pointers to both completed FVID2 Frames

So this results in the singe call to FVID2 Queue to submit frames related to all channels, and driver giving multiple callbacks to the application for the number of frames captured which results in the multiple de-queue calls for a single queue call.

Application can also opt to wait for the multiple callback and call FVID2 Queue which will return all the frames capture till then.

FVID2 ProcessFrames

An application calls this function to submit a video buffer to the video device driver. This API is very similar to FVID2 Queue API except that this work in M2M drivers. This function can be called from the task context unless driver specifies that it can be called from the interrupt context as well. This is a non blocking API unless driver specifies otherwise.

```
Int32 FVID2_processFrames(FVID2_Handle handle,
                          FVID2_ProcessList *processList);
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
processList - [OUT] Pointer to the FVID2 ProcessList structure containing the information about the FVID2 frame lists and frames that has to be queued to the driver for processing. **return** - FVID2_SOK on success, else appropriate FVID2 Error Code on failure.

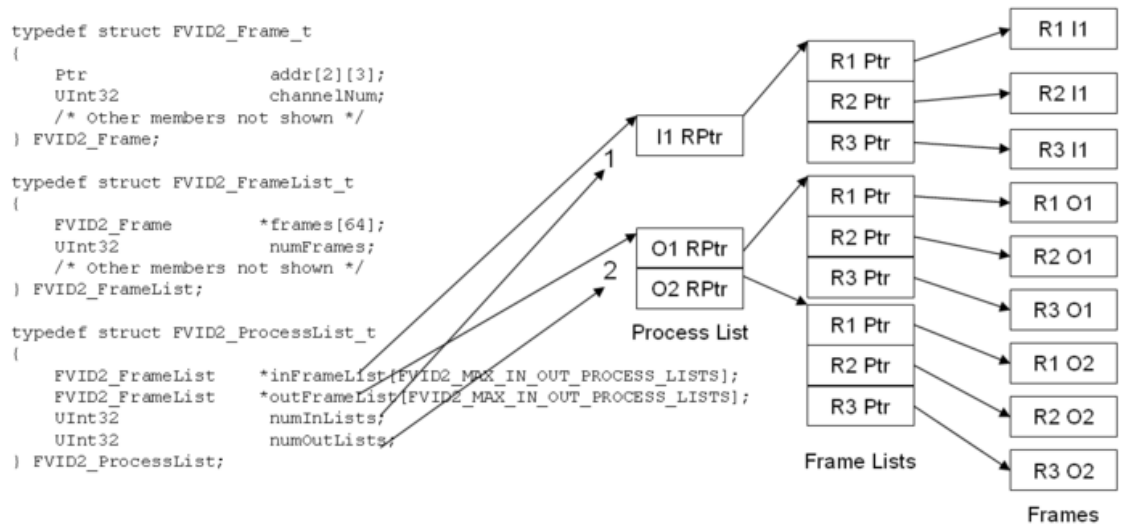
FVID2 GetProcessedFrames

An application calls this function to request the video device driver to give ownership of a video buffer.This API is very similar to the FVID2_dequeue API except that this is used in M2M drivers only. This is a non-blocking API if timeout is FVID2_TIMEOUT_NONE and could be called by task and ISR context unless the driver specifies otherwise. This is blocking API if timeout is FVID2_TIMEOUT_FOREVER if supported by specific driver implementation.

```
Int32 FVID2_getProcessedFrames(FVID2_Handle handle,
                               FVID2_ProcessList *processList,
                               UInt32 timeout);
```

handle - [IN] FVID2 handle returned by FVID2 Create call.
processList - [OUT] Pointer to the FVID2 ProcessList structure where the driver will copy the references to the dequeued FVID2 frame lists and frames.
timeout - [IN] FVID2 timeout. This will determine the timeout value till the driver will block for a free or completed buffer is available. For non-blocking drivers this parameter might be ignored.

Below figure shows how FVID2 FrameList pointers and FVID2 Frame pointers are initialized inside FVID2 ProcessList

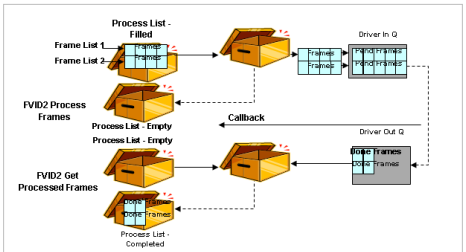


FVID2 ProcessList

As shown in above figure.

- Input Framelist pointer is initialized with one Framelist and output Framelist pointers are initialized with two FrameLists.
- So numInLists is set to 1 and numOutLists is set to 2
- In inputFramelist 3 Frame pointers are initialized with Frames.
- In outputFrameLists Frame pointers of both the framelists are intialized with three Frames each.

Below figure shows how FVID2 ProcessList are exchanged between the driver and application in the FVID2 ProcessFrames and FVID2 GetProcessedFrames APIs.



ProcessList Queue DeQueue

As show in figure FVID2_processFrames and FVID2_GetProcessedFrames is same like FVID2 Queue and FVID2 De-Queue in a single Queue and single De-Queue case except here the FVID2 ProcessList acts as containers instead of FVID2 FrameList

FVID2_getStandardInfo

Function to get the information about various FVID2 standards.Returns FVID2_SOK on success, else appropriate FVID2 error code on failure.

```
Int32 FVID2_getStandardInfo(FVID2_StandardInfo *stdInfo);
```

stdInfo - [OUT] Pointer to #FVID2_StandardInfo structure where the information is filled

FVID2 Error Codes

Following is the list of error codes that FVID2 APIs returns on successful or on the failure of the API. Each of the error codes is explained in the below code snapshot.

```
#define FVID2_SOK                       ((Int32) 0)
    /* FVID2 API call successful. */


#define FVID2_EFAIL                     ((Int32) -1)
    /*  FVID2 API call returned with error as failed. It may be some
     *  hardware failure or software failure */


#define FVID2_EBADARGS                  ((Int32) -2)
    /*  FVID2 API call returned with error as bad arguments. Typically
     *  NULL pointer passed to the FVID2 API where its not expected. */


#define FVID2_EINVALID_PARAMS           ((Int32) -3)
    /* FVID2 API call returned with error as invalid parameters.
Typically
     *  when parameters are not valid. */


#define FVID2_EDEVICE_INUSE             ((Int32) -4)
    /*  FVID2 API call returned with error as device already in use.
Tried
```

```
     *  to open the driver maximum + 1 times.  Display and Capture
driver suppports
     *  single open, while M2M driver supports multiple open. */


#define FVID2_ETIMEOUT                  ((Int32) -5)
    /*  FVID2 API call returned with error as timed out. Typically API
is
     *  waiting for some condition and returned as condition not
happened
     *  in the timeout period. */


#define FVID2_EALLOC                    ((Int32) -6)
    /*  FVID2 API call returned with error as allocation failure.
Typically
     *  memory or resource allocation failure. */


#define FVID2_EOUT_OF_RANGE             ((Int32) -7)
    /*  FVID2 API call returned with error as out of range. Typically
when
     *  API is called with some argument that is out of range for that
API like
     *  array index etc. */


#define FVID2_EAGAIN                    ((Int32) -8)
     /*  FVID2 API call returned with error as try again. Momentarily
API is
      *  not able to service request because of queue full or any other
 temporary
      *  reason. */


#define FVID2_EUNSUPPORTED_CMD          ((Int32) -9)
    /*  FVID2 API call returned with unsupported command. Typically
when
     *  command is not supported by control API. */


#define FVID2_ENO_MORE_BUFFERS          ((Int32) -10)
    /*  FVID2 API call returned with error as no more buffers
available.
     *  Typically when no buffers are available. */
```

```
#define FVID2_EUNSUPPORTED_OPS          ((Int32) -11)
    /*  FVID2 API call returned with error as unsupported operation.
     *  Typically when the specific operation is not supported by that
API such
     *  as IOCTL not supporting some specific functions. */


#define FVID2_EDRIVER_INUSE             ((Int32) -12)
    /* FVID2 API call returned with error as driver already in use. */
```

UserGuideHdvpssPlatformAPIs

Platform APIs and Drivers

Introduction

Platform APIs and driver does not fall into any of the FVID2 driver categories. These drivers and API are very much dependent on SoC and board. User may need to modify these APIs to suit their platform or board. Following is the list of platform APIs and their description.

FVID2_init

This is the first function to be called before calling any of the FVID2 APIs. It initializes all the data structures for FVID2 software stack. This is not a board dependent function. It doesn't require any change in case of board change. Internally different functions are called based on platform like DM814x, DM816x, DM8107 etc. So this function requires to be ported for all different platforms.

```
Int32 FVID2_init(Ptr args)
```

args - User should always pass `NULL` here.

return_val - Returns `FVID2_SOK` on success, else proper error code.

FVID2_deInit

This is the last function to be called after calling any of the FVID2 APIs. It De-initializes all the data structures initialized during FVID2_init.

```
Int32 FVID2_deInit(Ptr args)
```

args - User should always pass `NULL` here.

return_val - Returns `FVID2_SOK` on success, else proper error code.

Vps_platformInit

This is the platform initialization functions. It sets up the hardware for HDVPSS drivers. This function is platform dependent and it needs to be ported for different platforms like DM814x, DM816x, DM8107 etc. Function does following at at a high level for setting up of platform.

- Enabling of the HDVPSS functional clocks.
- Setting up of the display pixel clock for default values.
- Setting up of the pin mux for DVO2 in discrete sync mode with 24 data signals and 5 control signals.
- Setting up of the pin mux for VIP capture for 24/16 bit data signals and 5 control signals.
- Setting up of the I2C clocks for off-chip devices like TVPs and SILs
- Setting up of the interrupt muxing if required.

```
Int32 Vps_platformTI816xInit(Vps_PlatformInitParams *initParams)
```

initParams - Platform initialization parameters. Its explained below. **return_val** - Returns `FVID2_SOK` on success, else proper error code.

```
/**
 *  \brief Platform initialization parameters
 */
typedef struct
{
    UInt32  isPinMuxSettingReq;
    /**< Pinumx setting is requried or not. Sometimes pin mux setting
     *   is required to be done from Host operating system like Linux.
     */
} Vps_PlatformInitParams;
```

Vps_platformDeInit

This is platform De-Initialization function. It only clears the software states. Hardware states are maintained as is what was last before calling this function.

```
Int32 Vps_platformTI816xDeInit(void)
```

return_val - Returns `FVID2_SOK` on success, else proper error code. </pre>

Vps_platformDeviceInit

This is the initialization functions for all on-board devices like TVPs and SILs. This functions requires porting depending on platforms like DM814x, DM816x, DM8107 as well as on boards like Video Surveillance, Video Conferencing etc. It also requires change in case of the on-board devices is different or interfaced differently than what Video Surveillance and Video Conferencing boards supports. This function also initializes system driver, which provides APIs for setting up of pixel clock.

```
Vps_platformDeviceInit(Vps_PlatformDeviceInitParams *initPrms)
```

initParams - Platform device initialization parameters. Its explained below. **return_val** - Returns `FVID2_SOK` on success, else proper error code.

```
typedef struct
{
    UInt32  isI2cInitReq;
```

```
    /**< Indicates whether I2C initialization is required.
     * This is not requried in case all the on-board devices
     * are getting controlled by host operating system like Linux
     */

    UInt32  isI2cProbingReq;
    /**< If this is TRUE, Vps_platformDeviceInit will try to probe
all the I2C
     * devices connected on a specific I2C bus. This should be FALSE
for
     * all production  system since this is a time consuming function.
     * For debugging this should be kept TRUE to probe all on-board I2C
 devices.
     * This field is dont care if #isI2cInitReq=FALSE.
     */
} Vps_PlatformDeviceInitParams;
```

Vps_platformDeviceDeInit

This function De-initializes all external devices like filter, decoders and encoders. Deletion is only limited to deleting the software handles for all these devices. Devices states are actually not touched.

```
Vps_platformDeviceDeInit(void)
```

return_val - Returns `FVID2_SOK` on success, else proper error code.

System Driver

System driver provides application interface to configure Video display PLLs. Video display PLLs are shared between displays, so application needs to configure PLL based on the mode set on Venc. FVID2_create API is used to open the system driver. Plls can be set using the System driver ioctls with the handle got during FVID2_create.

```
FVID2_Handle FVID2_create(UInt32 drvId,
                          UInt32 instanceId,
                          Ptr createArgs,
                          Ptr createStatusArgs,
                          const FVID2_CbParams *cbParams);
```

</pre> **drvId** - `FVID2_VPS_VID_SYSTEM_DRV` System Driver ID. Use this ID to open system driver.

instanceId - `0`Instance ID is dont care for system controller driver.

createArgs - This parameters should be `NULL`

createStatusArgs - This parameter should be `NULL`.

cbParams - Since there is no callback from system driver, this parameters should be set to `NULL`.

FVID2_control

This API of system driver is used to expose control command provided by system driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

FVID2 Control - IOCTL_VPS_VID_SYSTEM_SET_VIDEO_PLL

Above control command is used to set PLL frequency of the selected Video encoder.

</pre> **handle** - Driver handle returned by create function call. This parameter should not be `NULL`.

cmd - `IOCTL_VPS_VID_SYSTEM_SET_VIDEO_PLL` ioctl.

cmdArgs - Pointer to `Vps_SystemVPllClk` structure containing venc on which PLL frequency needs to be set and the actual frequency that needs to be set. For details about structure please refer API guide.

cmdStatusArgs - This parameter should be `NULL`.

FVID2_delete

This API is used to closed the system driver handle previously opened.

```
Int32 FVID2_delete(FVID2_Handle handle, Ptr deleteArgs);
```

handle - Driver handle returned by create function call. This parameter should not be `NULL`. **deleteArgs - This parameter should be `NULL`.**

UserGuideHdvpssDisplayDriver

Display Drivers

Introduction

Display drivers takes the video buffer from the application and displays the video on the video encoder (VENC) at specified frame rate and resolution.

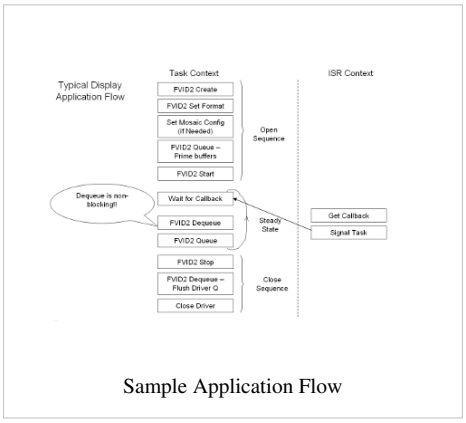
Display driver follows the FVID2 interface for the applications:

- Supports only one handle per instance. This means that a specific driver could be opened only once.
- Supports queuing mechanism. Application may queue multiple buffers with the driver and the driver displays the buffers one after the another sequentially in order the buffers are queued.
- Multiple buffer submission per queue/dequeue is not supported. Supports only one request per queue/dequeue operation. In order to queue/dequeue multiple buffers, the application has to call queue/dequeue multiple times.
- Queue and Dequeue FVID2 calls for all the display drivers are non blocking. However the control commands like programming of the scalar coefficients are blocking.
- Display driver calls the application call back function on displaying the application buffer. Application could dequeue the buffers by explicitly calling dequeue function after the callback.
- Once the display operation is started, the display driver always retains the last buffer and displays the same buffer continuously till the application gives a new buffer to display.
- Any dequeue call to get back the last buffer when display is in progress will return error. Application should stop display operation before it could dequeue the last buffer from the driver.

- When operating in interlaced mode, the display driver always takes both the field in a queue/dequeue call. The exception to this in de-interlace display driver where the driver works on a field at a time.
- Before the display operation is started, the application has to queue a minimum set of buffers. This operation is called priming.
- The minimum of number buffers required could defer from driver to driver. Generally this is equal to 1 buffer and the recommended value is equal to 3 buffers. Refer the driver specific documentation for the exact value.

Sample Application Flow

Following diagrams show the typical application flows for the display driver:



Sample Application Flow

Display Controller Driver

Introduction

This chapter describes the hardware overview, application software interfaces for Display Controller driver. The features and limitations of current driver implementation are listed in subsequent sections.

Important

The features supported or NOT supported in any release of the driver may vary from one HDVPSS driver release to another. See respective release notes for exact release specific details.

Features Supported

- Connecting multiplexers, VCOMP, CIG and COMP modules statically and dynamically (but not at run time, i.e. after display is started)
- Supports setting modes and synchronizing multiple VENCs
- Supports static configuration for VCOMP, CIG, EDE (TI816X only) and COMP modules of HDVPSS
- All HD VENCs support upto 720p60, 1080p30, 1080i60 and 1080p60 mode and SD VENC supports NTSC and PAL modes. Other modes are not supported.
- Supports FVID2 interface

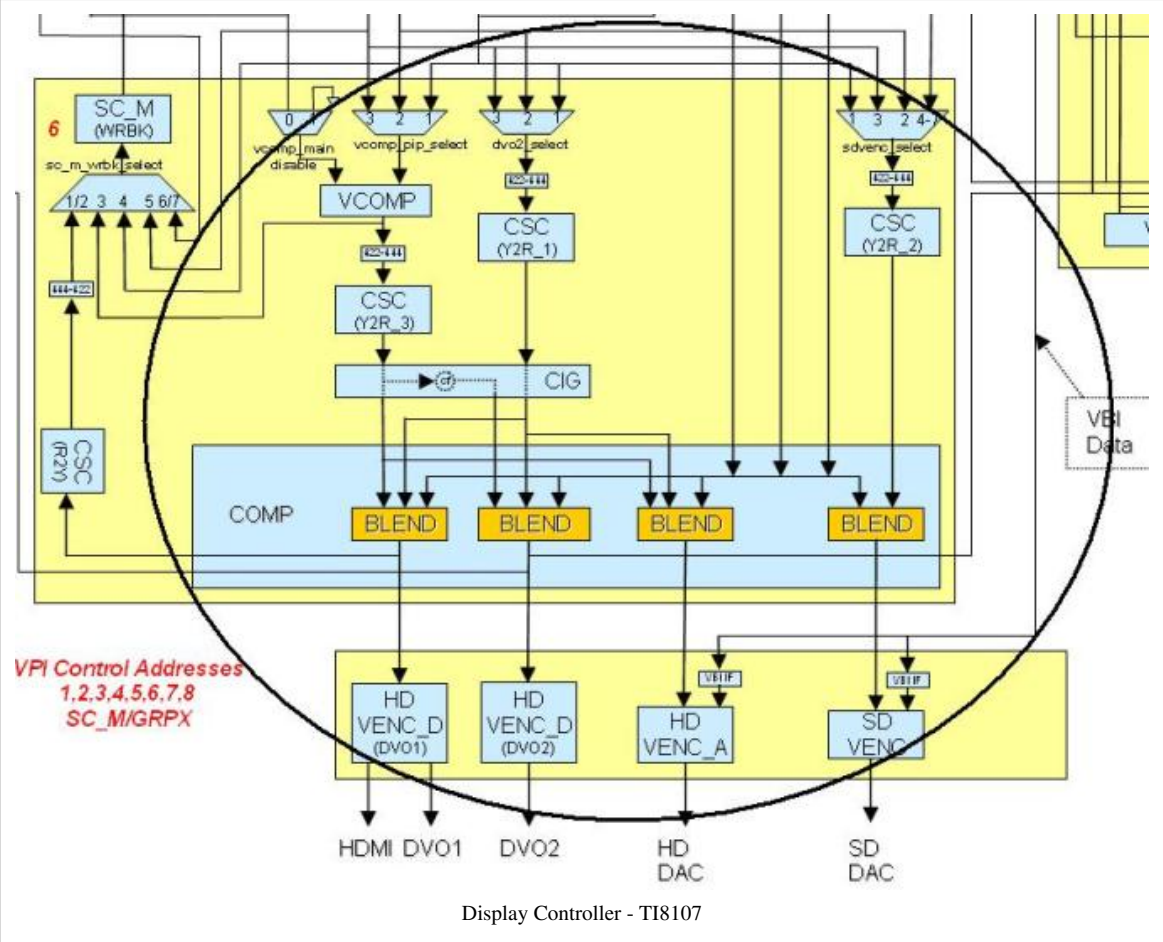
Features Not Supported

- Does not support configuring different modes on the tied VENCs like 1080P@60 FPS on DVO1 and 720P@60 FPS on DV02 could not be tied (synchronized)
- Run time configuration of VCOMP, CIG and blenders is not supported
- CPROC features are not supported. CPROC is currently put in simple bypass mode - does only color space conversion. Note that CPROC module is available only on TI816X.
- Run time switching of input path at the multiplexer and graphics enable/disable at the COMP is not supported.

Hardware Overview

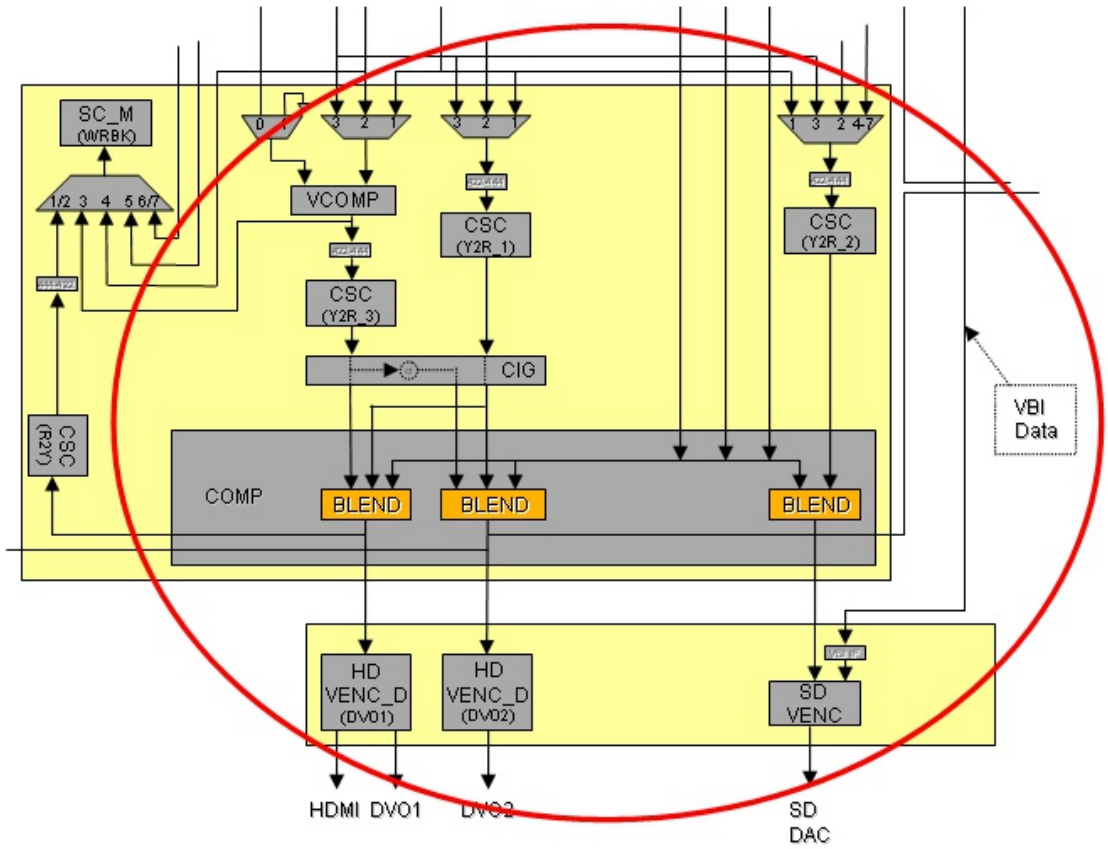
Below figures shows the complete HDVPSS Hardware. The circled part in the figure shows the modules which are controlled by display controller.

Overview - TI8107



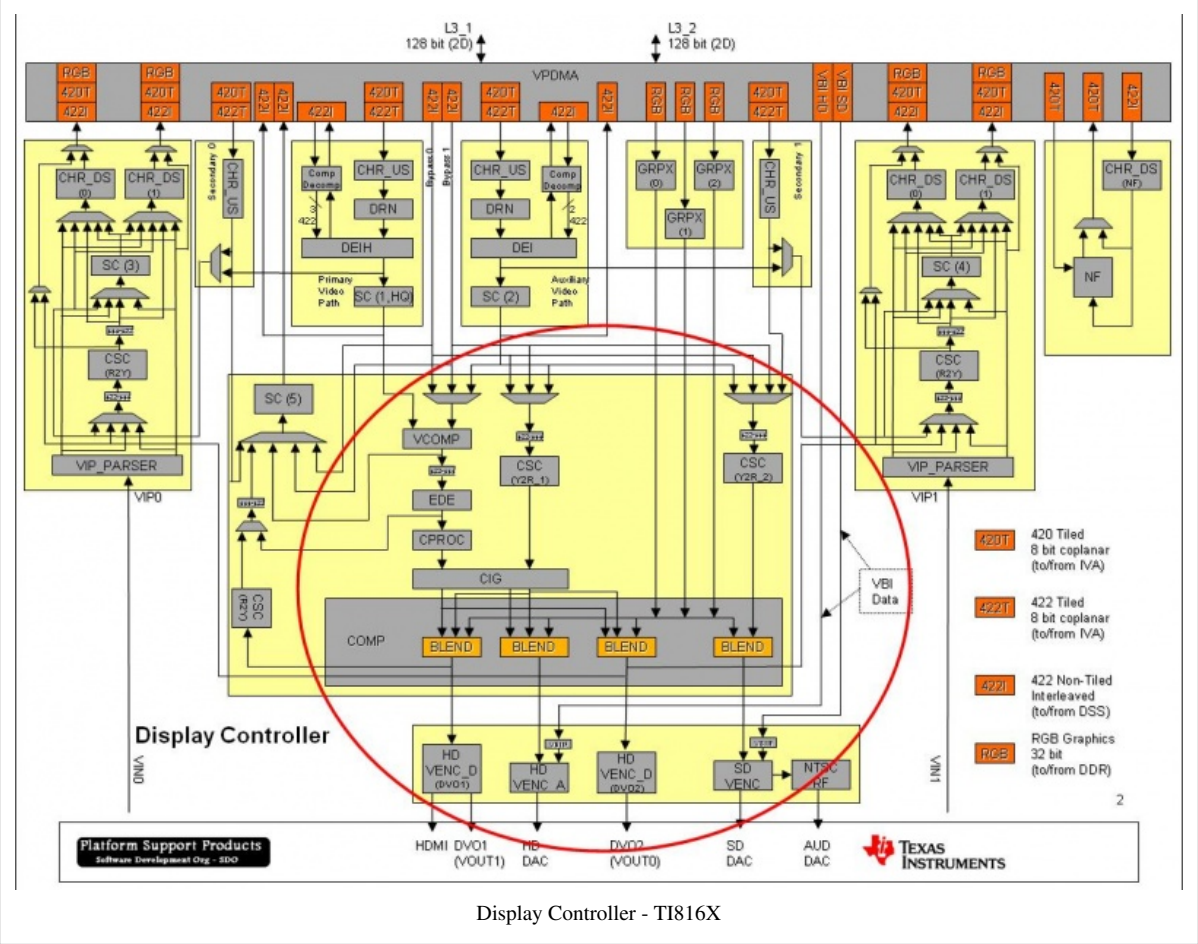
Display Controller - TI8107

Overview - TI814X



Display Controller - TI814X

Overview - TI816X



Display Controller - TI816X

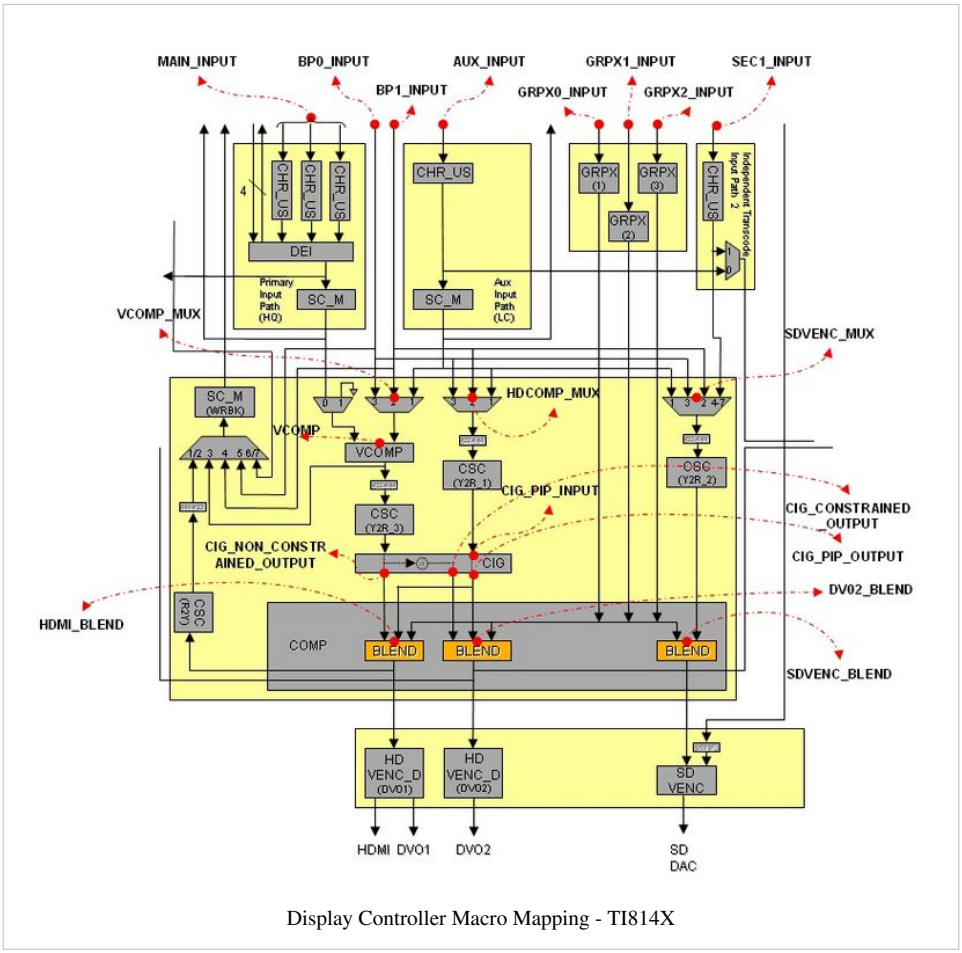
As shown in the figure, display driver controls configuration of VCOMP, CIG, EDE (TI816X only), CPROC (TI816X only), CSC, COMP and all the VENCs. It also configures the muxes to enable/disable the different paths to a particular VENC.

It provides APIs to the application to configure these paths and to set the different frame rates and resolutions in the VENC.

The display controller will provide necessary information to the display driver about the resolution and frame rate that it has to operate. This is abstracted from the application.

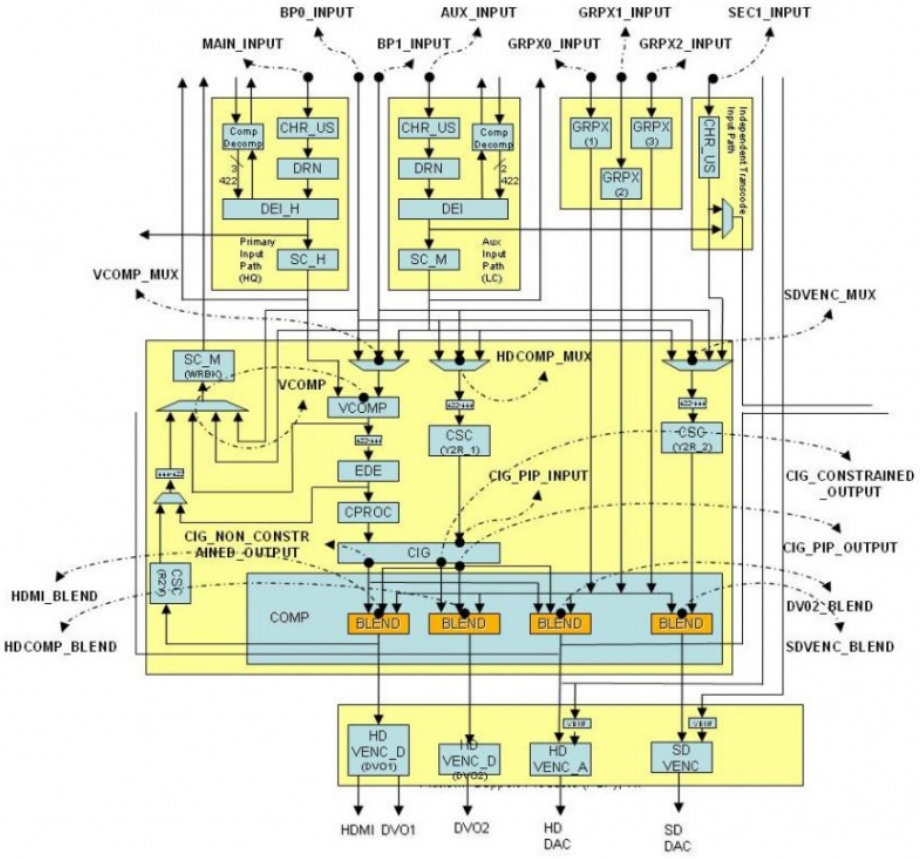
Below figures shows the mapping between the DCTRL nodes to the macro names as defined by the DCTRL interface file.

Macro mapping - TI814X



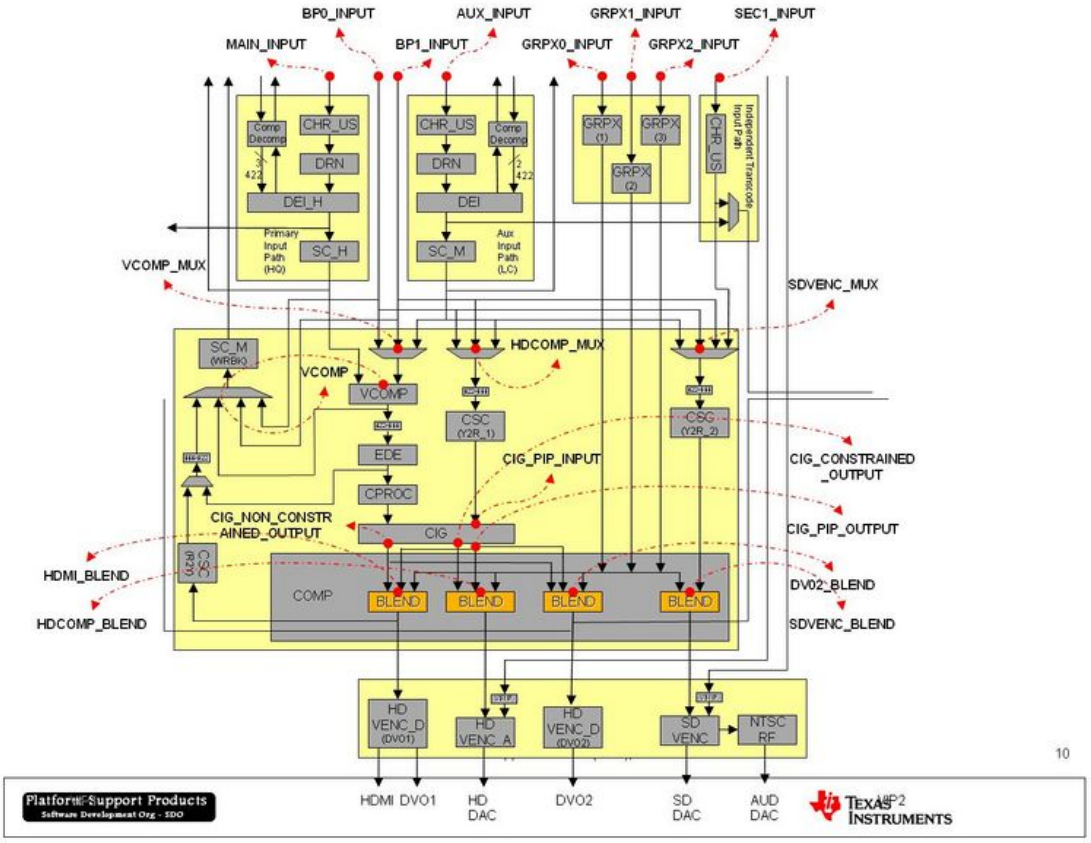
Display Controller Macro Mapping - TI814X

Macro mapping - TI8107



Display Controller Macro Mapping - TI8107

Macro mapping - TI816X



Display Controller Macro Mapping - TI816X

Software Application Interfaces

Display controller driver is not the streaming driver. Its used to control the specific part of the display controller as shown in above figure. The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: NA
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is de-initialized

The subsequent sections describe each phase in detail.

Note

Details of the structure, enumerations and #defines mentioned in the section can be found in HDVPSS API Guide

System Init Phase

The display driver sub-system initialization happens as part of overall HDVPSS system init. This API must be the the first API call before making any other FVID2 calls. Below section lists all the APIs which are part of the System Init phase.

FVID2 Init

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

Create Phase

In this phase user application opens or creates a driver instance. Any number of instances can be created for the display controller. Each instance works on the same central hardware block show above. Concurrency issues between the different handles is taken care by the display controller driver. User can pass number of parameters to the drivers during create phase like configuration of the path, settings of the venc etc, either through the create parameters or through the control commands.

FVID2 Create

This API is used to open the display controller driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

```
FVID2_Handle FVID2_create(UInt32 drvId,
                          UInt32 instanceId,
                          Ptr createArgs,
                          Ptr createStatusArgs,
                          const FVID2_CbParams *cbParams);
```

drvId - `FVID2_VPS_DCTRL_DRV` Display Controller Driver ID. Use this ID to open display controller driver. Details can be found in UserGuide

instanceId - `VPS_DCTRL_INST_0` Instance 0 of the display controller.

createArgs - Pointer to `Vps_DcCreateConfig` structure containing valid create params. This parameter can be NULL.

createStatusArgs - Pointer to `UInt32` return value where the driver returns the actual return code for create function. This parameter should not be NULL.

cbParams - Since there is no callback from the display controller, this parameters should be set to NULL.

FVID2 Control - Set Config

This is used to issue a control command to the driver. `IOCTL_VPS_DCTRL_SET_CONFIG` ioctl is used to set the entire display configuration in one shot. This ioctl takes pointer to the structure `Vps_DcConfig`. This structure takes either name of the use case or takes list of edges connecting nodes and configures display paths. It first validates these paths and then configures VPS for the display paths. It configures all display controller modules.

Important
This API should not be called after the display operation is started.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
```

```
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_DCTRL_SET_CONFIG` ioctl.

cmdArgs - Pointer to `Vps_DcConfig` structure containing Display Controller configuration. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Clear Config

This is used to issue a control command to the driver. `IOCTL_VPS_DCTRL_CLEAR_CONFIG` ioctl is used to clear the entire display configuration in one shot. This ioctl takes pointer to the structure `Vps_DcConfig`. This structure takes either name of the use case or takes list of edges connecting nodes and disables path between these nodes. It does not validate the edge list. It simply disables edge connecting nodes. For the vencs, it checks for the validity and then disables the VENC of there are no errors.

Important
This API should not be called after the display operation is started.

```
Int32 FVID2_control(FVID2_Handle handle,
                UInt32 cmd,
                Ptr cmdArgs,
                Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_DCTRL_CLEAR_CONFIG` ioctl.

cmdArgs - Pointer to `Vps_DcConfig` structure containing Display Controller configuration. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

Delete Phase

In this phase FVID2 delete API is called to close the driver instance. Remember to clear the configuration before closing the driver instance.

FVID2 Delete

This API is used to close the display controller driver. This is a blocking call and returns after closing the handle.

```
Int32 FVID2_delete(FVID2_Handle handle, Ptr deleteArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

deleteArgs - Not used currently. This parameter should be set to NULL.

System De-Init Phase

FVID2 de-Init

Display controller is de-initialized as a part of this phase. Here all resources acquired during system initialization are freed. Make sure all driver instances deleted before calling this API. Display sub-system de-init happens as part of overall FVID2 system de-init. Typically this is done during system shutdown.

```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

Sample Application

Refer specific Display driver sample examples which uses display controller functions to configure the paths and VENC settings.

Display Driver

Introduction

This chapter describes the hardware overview, application software interfaces, typical application flow and sample application usage for display driver involving bypass paths and secondary path.

The features and limitations of current driver implementation are listed in subsequent sections.

Important

Features Supported

Features Supported	Supported in TI816x	Supported in TI814x	Supported in TI8107
YUV422 interleaved format	YES	YES	YES
YUV420 and YUV422 semi planar format (only on SD path)	YES	NOT TESTED	YES
Interlaced and progressive displays	YES	YES	YES
Mosaic display support (only on Bypass paths)	YES	YES	YES
Dynamic mosaic layout change once the display is started (only on Bypass paths)	YES	YES	YES
Resolution upto 1080P@60FPS display on HD VENC D_DVO1/DVO2 through Bypass paths	YES	YES	YES
NTSC interlaced display on SD VENC through SD path	YES	NOT TESTED	YES
Non-blocking queue/dequeue operation	YES	YES	YES
Buffer from tiler (only on SD path)	YES	NOT TESTED	NOT TESTED
Periodic callback feature	YES	YES	YES
Runtime change of VCOMP cropping and positioning	YES	NOT TESTED	NOT TESTED
Field merged interlaced buffer mode	YES	YES	NOT TESTED
Field separated interlaced buffer mode	YES	YES	YES

Features Not Supported

- Runtime configuration of CIG and Blenders is not supported

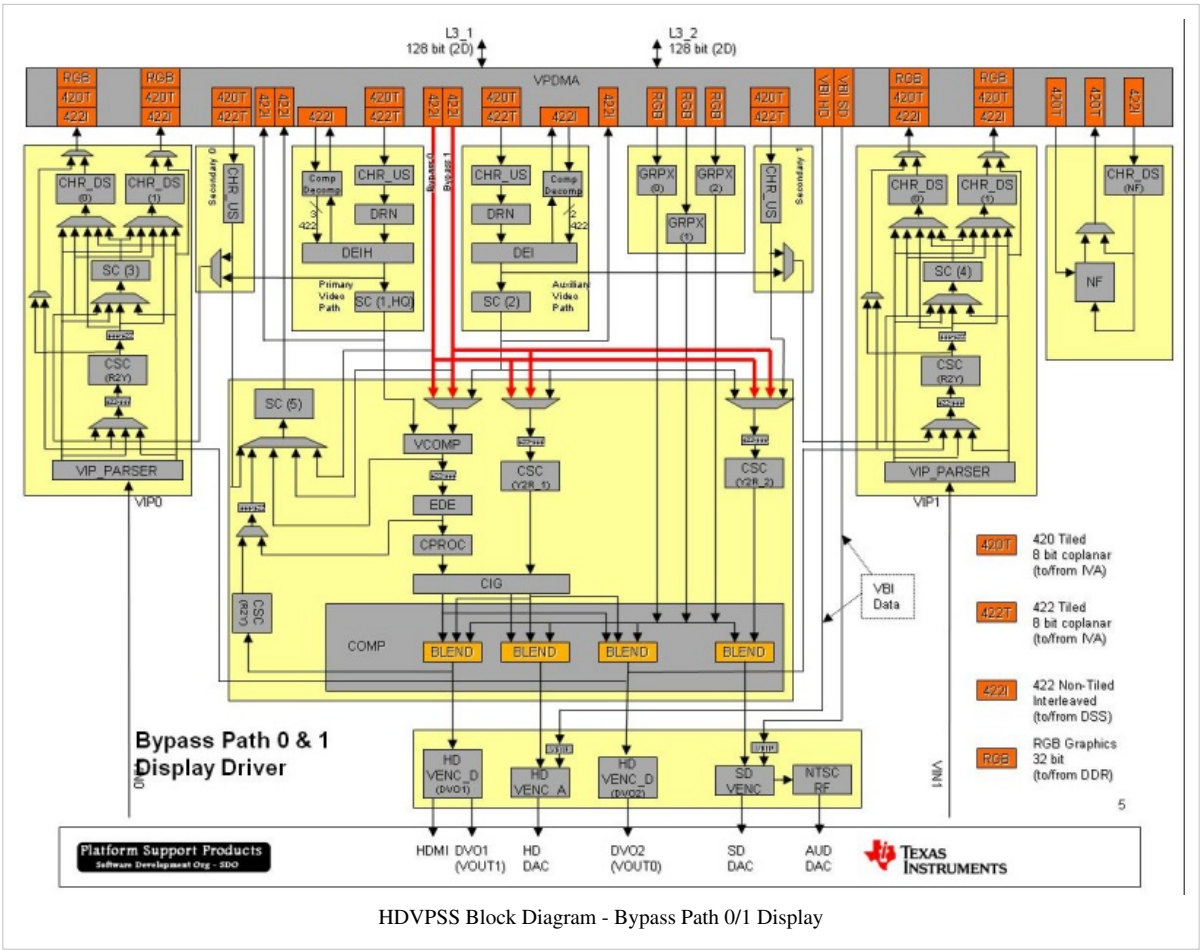
Features Supported

Following are the layouts tested through BP0 and BP1 paths for 1080p and 720p resolutions

- Full screen mode
- Non-Full screen mode
- 2x2 layout
- 4x4 layout
- 3x3 layout
- 8 Channel layout
- 6 Channel layout
- 2x1 layout

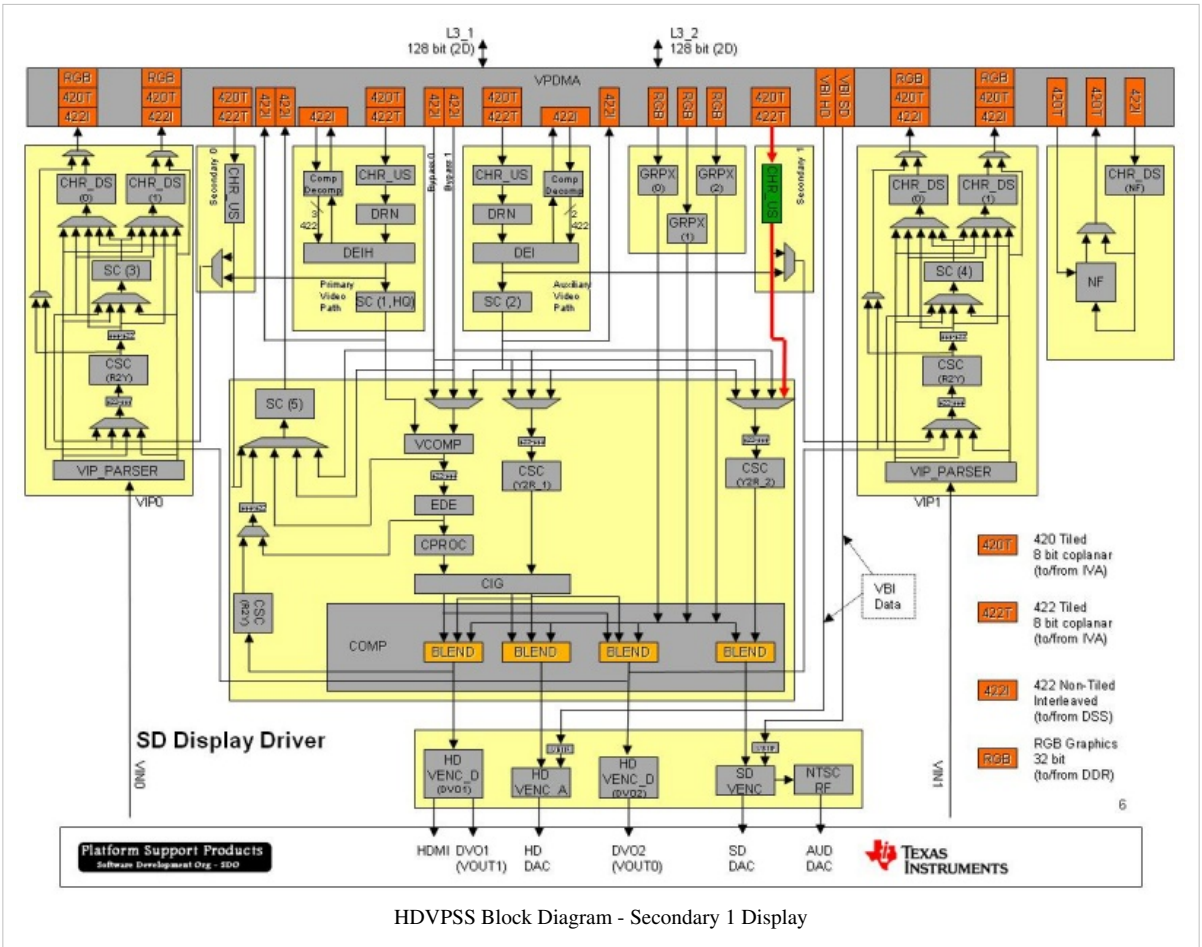
Hardware Overview

Below figures shows the complete HDVPSS Hardware. Two red bold lines in the figure shows the path on which the bypass path display driver operates.



HDVPSS Block Diagram - Bypass Path 0/1 Display

The red bold line on the right hand side of the image shows the secondary path display driver.



HDVPSS Block Diagram - Secondary 1 Display

As shown in the figure, display driver controls the three red line path in the hardware. It configures only up to the muxes. The rest of the hardware below the mux/switch like CIG, COMP, VENC etc are controlled by display controller driver. The display driver will communicate with the display controller internally to know about the resolution and frame rate that it has to operate. This is abstracted from the application.

Software Application Interfaces

The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: Here the driver is used to capture, process and release frames continuously
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is de-initialized

The subsequent sections describe each phase in detail.

Note

Details of the structure, enumerations and #defines mentioned in the section can be found in HDVPSS API Guide

System Init Phase

The display driver sub-system initialization happens as part of overall HDVPSS system init. This API must be the the first API call before making any other FVID2 calls. Below section lists all the APIs which are part of the System Init phase.

FVID2 Init

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

Create Phase

In this phase user application opens or creates a driver instance. Up to `VPS_DISPLAY_INST_MAX` (defined in vps_display.h) driver instances can be opened by a user. Each driver instance is associated with one of the bypass paths or the secondary path as listed in detail in HDVPSS API Guide.

User can pass number of parameters to the drivers during create phase like setting the format, setting the multi window configuration etc. These all configuration can be either done through standard FVID2 APIs or driver exported control commands or through driver create parameters itself.

FVID2 Create

This API is used to open the driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver. This cannot be called from ISR context.

```
FVID2_Handle FVID2_create(UInt32 drvId,
                          UInt32 instanceId,
                    Ptr createArgs,
                    Ptr createStatusArgs,
                    const FVID2_CbParams *cbParams);
```

drvId - `FVID2_VPS_DISP_DRV` to open the display driver.

instanceId - `VPS_DISP_INST_BP0` macro to open bypass path 0 display driver or pass `VPS_DISP_INST_BP1` macro to open bypass path 1 display driver or `VPS_DISP_INST_SEC1` macro to open secondary path display driver.

createArgs - Pointer to `Vps_DispCreateParams` structure containing valid create params. This parameter should not be NULL.

createStatusArgs - Pointer to `Vps_DispCreateStatus` structure containing the return value of create function and other driver information. This parameter could be NULL if application don't want the create status information.

cbParams - Pointer to `FVID2_CbParams` structure containing FVID2 callback parameters. This parameter should not be NULL. But the callback function pointers inside this structure is optional.

FVID2 Set Format

This API is used by the application to set the required buffer format for the display path. This is a blocking call and returns after setting the required format. This cannot be called from ISR context.

Note
This API should be called after the create function call to set the application buffer information. If the application fails to call this IOCTL, then the driver will assume buffer format according to the current VENC settings where this path is connected.

Note
When the application changes the VENC or any display controller settings after stopping the display driver, this IOCTL should be called before starting the display again. This ensures that the new display controller settings will be used by the driver. Otherwise the driver will work with the old information which could lead to issues.

Important
This API should not be called after the display operation has started.

```
Int32 FVID2_setFormat(FVID2_Handle handle, FVID2_Format *fmt);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

fmt - Pointer to `FVID2_Format` structure containing the format information. This parameter should not be NULL.

`channelNum` - Should be set to 0.

`width` - Frame width in pixels. This should be set less than or equal to the VENC settings.

`height` - Number of lines in the display frame. This should be set less than or equal to the VENC settings.

`pitch[FVID2_YUV_INT_ADDR_IDX]` - Should be atleast twice the input width in bytes in case of YUV422 interleaved format. `pitch[FVID2_YUV_SP_Y_ADDR_IDX]/pitch[FVID2_YUV_SP_CBCR_ADDR_IDX]` - should be atleast equal to the input width in bytes for YUV422/YUV420 semi-planar formats.

`scanFormat` - `FVID2_SF_INTERLACED` or `FVID2_SF_PROGRESSIVE`.

`fieldMerged[FVID2_YUV_INT_ADDR_IDX]` or `fieldMerged[FVID2_YUV_SP_Y_ADDR_IDX]` or `fieldMerged[FVID2_YUV_SP_CBCR_ADDR_IDX]` - For interlaced display TRUE if fields are merged, FALSE if fields are separated. For progressive display this should be set to FALSE.

`dataFormat` - `FVID2_DF_YUV422I_YUYV` for bypass path and secondary path, `FVID2_DF_YUV420SP_UV` or `FVID2_DF_YUV422SP_UV` for secondary path.

`bpp` - `FVID2_BPP_BITS16` for YUV422 format or `FVID2_BPP_BITS12` for YUV420 format.

FVID2 Get Format

This API is used by the application to get the current buffer format for the display path. This is a blocking call and returns after getting the required format. This cannot be called from ISR context.

```
Int32 FVID2_getFormat(FVID2_Handle handle, FVID2_Format *fmt);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

fmt - Pointer to `FVID2_Format` structure where the format information needs to be copied by the driver. This parameter should not be NULL.

FVID2_control

This API is used to expose the different control commands to the application depending upon the specific driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

FVID2 Control - Create Layout

`IOCTL_VPS_CREATE_LAYOUT` ioctl is used to create a layout depending on the multiple window parameter. This is a blocking call. This IOCTL creates the necessary infrastructure for the specified layout. The user has to call select multiple window layout IOCTL to explicitly select a particular layout before starting display. This cannot be called from ISR context. This API is not supported for the secondary path display driver instance.

Note
This API could be called after the display operation has started.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_CREATE_LAYOUT` ioctl.

cmdArgs - Pointer to `Vps_MultiWinParams` structure containing valid multiple window parameters. This parameter should not be NULL. For the bypass path display driver, different data format and BPP for each individual windows is not supported. They should be set to `FVID2_DF_YUV422I_YUYV` and `FVID2_BPP_BITS16` respectively. When windows overlap, `priority` should be set depending on which window should be displayed.

cmdStatusArgs - Pointer to `Vps_LayoutId` structure containing the unique layout ID to be used by the application for future reference. This parameter should not be NULL.

FVID2 Control - Delete Layout

`IOCTL_VPS_DELETE_LAYOUT` ioctl is used to delete an already created layout. This is a blocking call. This cannot be called from ISR context. When the layout to delete is used by the current display or is not created, this returns error. This API is not supported for the secondary path display driver instance.

Note
All the created layouts will be automatically deleted (if not deleted by this IOCTL) when the application closes the driver.

Note
This API could be called after the display operation has started.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_DELETE_LAYOUT` ioctl.

cmdArgs - Pointer to `Vps_LayoutId` structure containing delete parameter containing the layout ID to delete. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Select Layout

`IOCTL_VPS_SELECT_LAYOUT` ioctl is used to select an already created layout for display. This is a blocking call. This cannot be called from ISR context. This API is not supported for the secondary path display driver instance.

Note
This IOCTL should be called before starting the display to select the default layout to start with. This IOCTL could not be called once the display starts. For changing the layout after operation is started, application could do so by passing the layout ID (which is returned through create layout IOCTL) as a part of display runtime parameter `Vps_DispRtParams` which needs to be passed along with frame list `perListCfg`.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SELECT_LAYOUT` ioctl.

cmdArgs - Pointer to `Vps_LayoutId` structure containing select parameter containing the layout ID to select. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Delete All Layout

`IOCTL_VPS_DELETE_ALL_LAYOUT` ioctl is used to delete all the created layouts. Typically this is used after stopping the display and before closing the driver. This is a blocking call. This cannot be called from ISR context. This IOCTL could not be called when display is in progress with one of the created layout. This API is not supported for the secondary path display driver instance.

Note
All the created layouts will be automatically deleted (if not deleted by this IOCTL) when the application closes the driver.

Note
This API could be called after the display operation has started.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_DELETE_ALL_LAYOUT` ioctl.

cmdArgs - Not used currently. This parameter should be set to NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

Run Phase

This phase is used to start or stop the already started display driver. This is also used to exchange the displayed buffers and the fresh buffers between the driver and applications.

FVID2 Start

This API is used by the application to start the display operation. This is a blocking call and returns after starting the display operation. Before starting the display operation, the application has to prime at least 1 buffer with the driver using queue API. Typically 3 buffers are used: 1 used by application and 2 buffers are queued with the driver at any given time. This cannot be called from ISR context.

```
Int32 FVID2_start(FVID2_Handle handle, Ptr cmdArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmdArgs - Not used currently. This parameter should be set to NULL.

FVID2 Stop

This API is used by the application to stop the display operation. This is a blocking call and returns after stopping the display operation. This cannot be called from ISR context.

```
Int32 FVID2_stop(FVID2_Handle handle, Ptr cmdArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmdArgs - Not used currently. This parameter should be set to NULL.

FVID2 Queue

This API is used to submit a video buffer to the driver for display operation. This is a non-blocking call and could be called from task or ISR context. Once the buffer is queued the application loses ownership of the buffer and is not suppose to modify or use the buffer.

```
Int32 FVID2_queue(FVID2_Handle handle,
                  FVID2_FrameList *frameList,
                  UInt32 streamId);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

frameList - Pointer to `FVID2_FrameList` structure containing the pointer to the FVID2 frames. This parameter should not be NULL. In normal display operation, the number of frames passed using this call is one. When multiple window configuration is set, this frame list contains the frames of all the multiple window buffers representing a single composited video frame. For queuing multiple window buffers, the frames should be given in the same order in which the window parameters are specified while creating a layout using `IOCTL_VPS_CREATE_LAYOUT` ioctl.

streamId - Not used currently. This parameter should be set to 0.

FVID2 Dequeue

This API is used by the application to get ownership of a displayed video buffer from the display driver. This is a non-blocking call and could be called from task or ISR context.

```
Int32 FVID2_dequeue(FVID2_Handle handle,
                    FVID2_FrameList *frameList,
                    UInt32 streamId,
                    UInt32 timeout);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

frameList - Pointer to `FVID2_FrameList` structure where the driver will copy the displayed FVID2 frames. This parameter should not be NULL. In normal display operation, the number of frames returned using this call is one. When multiple window configuration is set, this frame list returns the frames of all the multiple window buffers of the current layout.

streamId - Not used currently. This parameter should be set to 0.

timeout - Not used currently as only non-blocking queue/dequeue operation is supported. This parameter should be set to FVID2_TIMEOUT_NONE.

Delete Phase

In this phase FVID2 delete API is called to close the driver instance. All the resources are freed. Make sure display is stopped using FVID2_stop() before deleting a display instance. Once the driver is instance is closed it can be opened again with new configuration.

FVID2 Delete

This API is used to close the display driver. This is a blocking call and returns after closing the handle. This cannot be called from ISR context.

Note
Closing the driver will implicitly stop the display if stop IOCTl is not called by the application. This will also delete all the created layouts.

```
Int32 FVID2_delete(FVID2_Handle handle, Ptr deleteArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

deleteArgs - Not used currently. This parameter should be set to NULL.

System De-Init Phase

FVID2 de-Init

In this phase display driver is de-initialized. Here all resources acquired during system initialization are free'ed. Make sure all driver instances deleted before calling this API. Display sub-system de-init happens as part of overall FVID2 system de-init. Typically this is done during system shutdown.

```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

Sample Application

Mosaic Display

This example illustrates the dynamic mosaic feature of Bypass Path 0 by displaying different layouts on On-Chip HDMI/HDDAC/Off-Chip HDMI/LCD (Present in VS/VC/VCAM daughter cards) outputs.
Since the input buffer size used for the mosaic windows is same, this application makes use of the buffer pitch to display a smaller window by cropping the input buffer according to the layout. This application creates all possible combinations of 4x4 layouts like, 1x1, 1x2, 1x3, 1x4, 2x1 so on and up to 4x4 layouts. Then while the display is in progress, changes the layout every `LAYOUT_SWITCH_RATE` frames.
For On-Chip HDMI display, the On-Chip HDMI encoder should be configured from the A8 core after running this

example using the executable provided along with this package at *$(rel_folder)\docs\ti816x\TI816x_A8_HDMI_Sample.out* for TI816x or *$(rel_folder)\docs\ti814x\TI814x_A8_HDMI_Sample.out* for TI814x/TI8107.

- Please refer Common Steps for connecting CCS to TI81xx, running gel file etc.
- Load *hdvpss_examples_mosaicDisplay.xem3* executable file found at *$(rel_folder)\build\example-name\bin\$platform\example-name-whole-program-debug.xem3* to DSS M3 debug session
- Run the application and it will halt for the user to provide the desired VENC and then the desired VENC resolution.
- Then it will halt for the user to load the input frames.
- Using loadRaw command in script console of CCS, load 5 frames of 1920 x 1080 YUYV interleaved data to the printed location. (Ignore "syntax error" if it appears during loading)

```
loadRaw(<Address Location>, 0, " < File Path > ", 32,
false);
```

- Enter 1 on the console when application stops at after loading of the frames is completed.

```
Input a numeric key and press enter after loading...
```

- This will display the various layouts one after the other on the video probe. By default, the layout will switch every 6 seconds for 1080p60/720p60 or for every 12 seconds for 1080p30/1080i60. It could be changed by changing the macro `LAYOUT_SWITCH_RATE` to any value from 1.
- Application will stop after displaying `TOTAL_LOOP_COUNT` frames
- Configuration Options: To display in bypass path 1, change the macro `DRIVER_INSTANCE` to `VPS_DISP_INST_BP1`
- Configuration Options: To change the number of frames to display, change the macro `TOTAL_LOOP_COUNT` to any desired value greater than 2.
- Configuration Options: Change the `LAYOUT_SWITCH_RATE` macro to change how often the layout should change. To disable dynamic layout change, set this to more than the application loop count.
- Configuration Options: Change `BUFFER_WIDTH` and `BUFFER_HEIGHT` macro according to the input buffer dimension. The application will automatically change the layout window sizes according to the buffer size and pitch.
- This application also displays WSVGA @70fps on VCAM LCD

NTSC Display on SD VENC

This example illustrates secondary path SD display by displaying a 720 x 480 image on SD VENC configured for NTSC.

- Please refer Common Steps for connecting ccs to TI816x, running gel file etc.
- Load *hdvpss_examples_sdDisplay.xem3* executable file found at *$(rel_folder)\build\example-name\bin\$platform\example-name-whole-program-debug.xem3* to DSS M3 debug session
- Run the application and it will halt for the user to load the input frames.
- Using loadRaw command in script console of CCS, load 10 frames of 720 x 480 YUV420 Semiplanar (Y in one plane, UV interleaved in other plane)

data to location as printed on the console. (Ignore "syntax error" if it appears during loading)

```
loadRaw(<Location>, 0, " < File Path > ", 32, false);
```

- Enter 1 on the console when application stops at after loading of the frames is completed.

```
Input a numeric key and press enter after loading...
```

- This will display video on the SD VENC.
- Application will stop after displaying 3000 frames
- Configuration Options: To change the number of frames to display, change the macro `TOTAL_LOOP_COUNT` to any desired value greater than 2

Tripple Display

- Please refer Common Steps for connecting ccs to TI816x, running gel file etc.
- Load *hdvpss_examples_triDisplay.xem3* executable file found at *$$(rel_folder)\build\example-name\bin\$platform\example-name-whole-program-debug.xem3* to DSS M3 debug session
- Run the application and it will halt for the user to select the desired Display Combo.
- It will also half for the user to load input images
- Using loadRaw command in script console of CCS, load 10 frames of 720 x 480 YUV422 Interleaved.

data to location as printed on the console. (Ignore "syntax error" if it appears during loading)

```
loadRaw(<Location>, 0, " < File Path > ", 32, false);
```

- Enter 1 on the console when application stops at after loading of the frames is completed.

```
Input a numeric key and press enter after loading...
```

- Application uses same input images to display on all active displays.
- This will display video either on on-chip HDMI, on DVO2 or on SD VENC or combination of these three display.
- Application will stop after displaying 1000 frames on all active displays.
- Configuration Options: To change the number of frames to display, change the macro `TOTAL_LOOP_COUNT` to any desired value greater than 2

Sample Application with on-chip HDMI

Following is the procedure for setting the output to the on-chip HDMI and running the on-chip HDMI sample example on A8.

'TI816X'

- Download the "$HDVPSS_install_dir\pspdrivers_\docs\ti816x\TI816x_A8_HDMI_Sample.out to A8 after running the gel option specified in respective applications.
- Run HDMI sample with desired resolution.
- Run the respective sample application on M3.

'TI814X/TI8107'

- Run the A8 gel file as required by the respective sample application.
- Run the M3 gel file for enabling the cache as specified in respective application.
- Load "$HDVPSS_install_dir\pspdrivers_\docs\ti814x\TI814x_A8_HDMI_Sample.out to A8 at any point when M3 code halts for console input (scanf).
- Run TI814x_A8_HDMI_Sample.out with desired resolution.
- Run the gel file on A8 with "HDMI_PLL_Config_1_485_GHz" or "HDMI_PLL_Config_742_5_MHz" option based on resolution selected. This is to be done from gel file because the A8 .out for the HDMI configuration runs in "User Mode" and PRCM configuration is not allowed from "User Mode" of the processor.
- If using Ccsv5 halt M3 processor while giving input on A8 and vice versa.
- Now continue the M3 application and select on-chip HDMI as the option.

These steps remain same for all the applications which requries the on-chip HDMI as one of the display. HDMI.out provided with this release will be removed from next release onwards. It will be demonstrated on how to use HDMI kernel module to run tri-display application.

Graphics Path 0/1/2 Display Driver

Introduction

This chapter describes the hardware overview, application software interfaces, typical application flow and sample application usage for Graphics path 0/1/2 display driver. The features and limitations of current driver implementation are listed in subsequent sections.

Important
The features supported or NOT supported in any release of the driver may vary from one HDVPSS driver release to another. See respective release notes for exact release specific details.

Features Supported

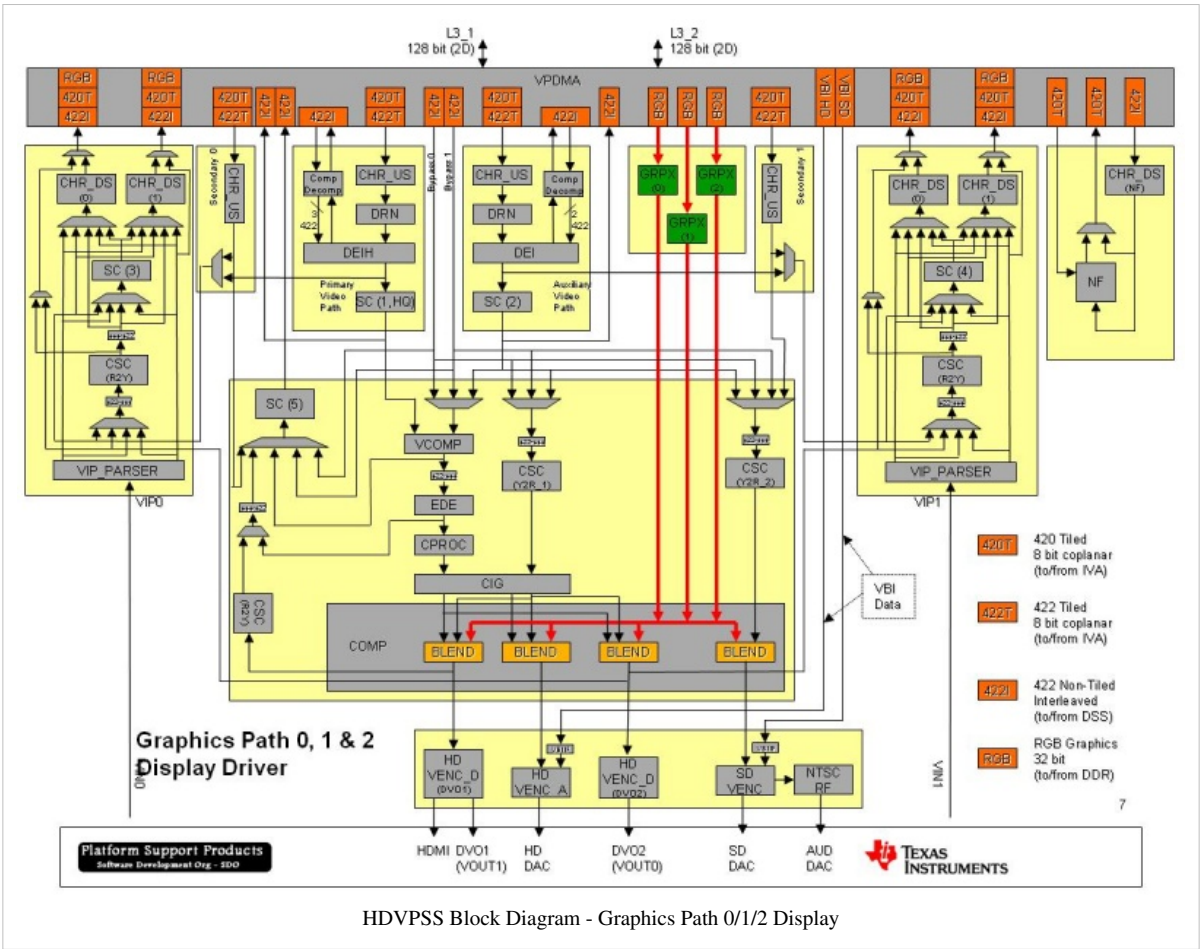
- Supports display of the following graphics data format via Graphics paht 0/1/2
- RGB16-565
- ARGB16-1555
- ARGB16-4444
- RGB16A-5551
- RGBA16-4444
- ARGB24-6666
- RGB24-888
- ARGB32-8888
- RGBA24-6666
- RGBA32-8888
- various bitmap 8/4/2/1 bit
- Supports both interlaced and progressive graphics data
- Supports FVID2 streaming model - queue and de-queue of the buffers
- Supports FVID2 non-streaming model - frame buffer mode
- multiple region display support - Supports up to 12 region display on a single frame
- Supports 1080P@60FPS progressive display on HD VENC D_DVO1/DVO2 and NTSC interlaced display on SD VENC
- Supports non-blocking queue/dequeue operation
- Supports runtime region feature changes: position, dimension, alpha blending, boundbox blending, stenciling, tranparency masking and scaling.
- Supports runtime scaling ratio changes among frames.

Features Not Supported

- Multiple region use case is not supported when operated under frame buffer mode
- FVID2 error callback feature is not supported
- Changing scaling ratio between regions in any givien display frame is not supported
- Stenciling is not supported on TI814x.

Hardware Overview

Below figures shows the complete HDVPSS Hardware.



HDVPSS Block Diagram - Graphics Path 0/1/2 Display

The red bold lines in the figure shows the path on which the graphics path display driver operates. As shown in the figure, graphics path display driver controls the three red line path in the hardware. It configures only itself. The rest of the hardware below like COMP, VENC etc are controlled by display controller driver. The display driver will communicate with the display controller internally to know about the resolution and frame rate that it has to operate. This is abstracted from the application. Please refer to the HDVPSS Graphics hardware specification for details information.

Software Application Interfaces

The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: Here the driver is used to capture, process and release frames continuously
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is de-initialized

The subsequent sections describe each phase in detail.

Note
Details of the structure, enumerations and #defines mentioned in the section can be found in HDVPSS API Guide

System Init Phase

The display driver sub-system initialization happens as part of overall HDVPSS system init. This API must be the the first API call before making any other FVID2 calls. Below section lists all the APIs which are part of the System Init phase.

FVID2 Init

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

Create Phase

In this phase user application opens or creates a driver instance. Up to VPS_DISP_GRPX_MAX_INST (defined in vps_graphics.h) driver instances can be opened by a user. Each driver instance is associated with one of the graphics path.

User can pass number of parameters to the drivers during create phase like setting the format, setting the multi region configuration etc. These all configuration can be either done through standard FVID2 APIs or driver exported control commands or through driver create parameters itself.

FVID2 Create

This API is used to open the driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

```
FVID2_Handle FVID2_create(UInt32 drvId,
                  UInt32 instanceId,
                  Ptr createArgs,
                  Ptr createStatusArgs,
                  const FVID2_CbParams *cbParams);
```

drvId - `FVID2_VPS_DISP_GRPX_DRV` to open the display driver.

instanceId - `VPS_DISP_INST_GRPX0` macro to open graphics path 0 display driver or pass `VPS_DISP_INST_GRPX1` macro to open graphics path 1 display driver or pass `VPS_DISP_INST_GRPX2` macro to open graphics path 2 display driver.

createArgs - Pointer to `Vps_GrpxCreateParams` structure containing valid create params. This parameter should not be NULL.

createStatusArgs - Pointer to `Vps_GrpxCreateStatus` structure containing the return value of create function and other driver information. This parameter could be NULL if application does not want the status information.

cbParams - Pointer to `FVID2_CbParams` structure containing FVID2 callback parameters. This parameter should not be NULL. But the callback function pointers inside this structure is optional.

FVID2 Set Format

This API is used by the application to set the required buffer format for the display path. This is a blocking call and returns after setting the required format.

Note
This API should be called immediately after the create function call to set buffer information. if application failed to call this IOCTL, driver will return error when other IOCTls are called..

Note
When the application changes the VENC settings after stopping the display driver, this IOCTL should be called before starting the display again. This ensures that the new display controller settings will be used by the driver. Otherwise the driver will work with the old information which could lead to issues.

Important
This API should not be called after the display operation is started.

```
Int32 FVID2_setFormat(FVID2_Handle handle, FVID2_Format *fmt);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

fmt - Pointer to `FVID2_Format` structure containing the format information. This parameter should not be NULL. Below are the supported formats for graphics path display driver.

`channelNum` - Should be set to 0.

`width` - Region width in pixels.

`height` - Number of lines in the display region.

`pitch[FVID2_YUV_INT_ADDR_IDX]` - in bytes, this is up to the graphics data format.

`scanFormat` - `FVID2_SF_INTERLACED` or

`FVID2_SF_PROGRESSIVE`, this is graphics data format instead of display format, most of time, this shall be set to `FVID2_SF_PROGRESSIVE`.

`fieldMerged[FVID2_YUV_INT_ADDR_IDX]` - For interlaced graphics input data, TRUE if fields are merged, FALSE if fields are separated. For progressive graphics input this should be set to FALSE.

`dataFormat` - `FVID2_DF_RGB24_888` , `FVID2_DF_ARGB16_1555` `FVID2_DF_RGBA16_5551` , `FVID2_DF_ARGB16_4444` `FVID2_DF_RGBA16_4444` , `FVID2_DF_ARGB24_6666` `FVID2_DF_RGBA24_6666` , `FVID2_DF_RGBA32_8888` `FVID2_DF_ARGB32_8888` , `FVID2_DF_RGB16_565` `FVID2_DF_BITMAP4_LOWER` , `FVID2_DF_BITMAP4_UPPER` `FVID2_DF_BITMAP2_OFFSET0` , `FVID2_DF_BITMAP2_OFFSET1` `FVID2_DF_BITMAP2_OFFSET2` , `FVID2_DF_BITMAP2_OFFSET3` `FVID2_DF_BITMAP1_OFFSET0` , `FVID2_DF_BITMAP1_OFFSET1` `FVID2_DF_BITMAP1_OFFSET2` , `FVID2_DF_BITMAP1_OFFSET3` `FVID2_DF_BITMAP1_OFFSET4` , `FVID2_DF_BITMAP1_OFFSET5` `FVID2_DF_BITMAP1_OFFSET6` , `FVID2_DF_BITMAP1_OFFSET7` `FVID2_DF_BITMAP8`.

`bpp` - based on `dataFormat`, `FVID2_BPP_BITS1`, `FVID2_BPP_BITS2`, `FVID2_BPP_BITS4`, `FVID2_BPP_BITS8`, `FVID2_BPP_BITS16`, `FVID2_BPP_BITS24`, `FVID2_BPP_BITS32`.

FVID2 Get Format

This API is used by the application to get the buffer format set for the display path. This is a blocking call and returns after getting the required format.

```
Int32 FVID2_getFormat(FVID2_Handle handle, FVID2_Format *fmt);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

fmt - Pointer to `FVID2_Format` structure where the format information needs to be copied by the driver. This parameter should not be NULL.

FVID2 Control - Create Multiple Regions Layout

This is used to issue a control command to the driver. `IOCTL_VPS_CREATE_LAYOUT` ioctl is used to create a multiple region layout depending on the multiple window(region) parameters. This is a blocking call. This call can not be called from ISR context.

Important
This API should not be called after the display operation is started.

```
Int32 FVID2_control(FVID2_Handle handle,
            UInt32 cmd,
            Ptr cmdArgs,
            Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_CREATE_LAYOUT` ioctl.

cmdArgs - Pointer to `Vps_MultiWinParams` structure containing valid multiple window(region) parameters. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Set Graphcis Regions Parameters

This is used to issue a control command to the driver. `IOCTL_VPS_SET_GRPX_PARAMS` ioctl is used to set the graphics regions parameters. This is a blocking call.

Important
This API should not be called after the display operation is started.

```
Int32 FVID2_control(FVID2_Handle handle,
            UInt32 cmd,
            Ptr cmdArgs,
            Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SET_GRPX_PARAMS` ioctl.

cmdArgs - Pointer to `Vps_GrpxParamsList` structure containing valid graphics region parameters. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Get Graphcis Regions Parameters

This is used to issue a control command to the driver. `IOCTL_VPS_GET_GRPX_PARAMS` ioctl is used to get the graphics regions parameters. This is a blocking call.

Important
This API could be called after the display operation is started.

```
Int32 FVID2_control(FVID2_Handle handle,
            UInt32 cmd,
            Ptr cmdArgs,
            Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_GET_GRPX_PARAMS` ioctl.

cmdArgs - Pointer to `Vps_GrpxParamsList` structure containing valid graphics region parameters. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

Run Phase

This phase is used to start or stop the already started display driver. This is also used to exchange the displayed buffers and the fresh buffers between the driver and applications.

FVID2 Start

This API is used by the application to start the display operation. This is a blocking call and returns after starting the display operation. Before starting the display operation, the application has to prime at least 1 buffer with the driver using queue API. When driver is openned under `VPS_GRPX_NON_FRAME_BUFFER_MODE`, typically 3 buffers are used - 1 used by application and 2 buffers are queued with the driver at any given time. When driver is openned under `VPS_GRPX_FRAME_BUFFER_MODE`, typically 1 buffer is used.

```
Int32 FVID2_start(FVID2_Handle handle, Ptr cmdArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmdArgs - Not used currently. This parameter should be set to NULL.

FVID2 Stop

This API is used by the application to stop the display operation. This is a blocking call and returns after stopping the display operation.

```
Int32 FVID2_stop(FVID2_Handle handle, Ptr cmdArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmdArgs - Not used currently. This parameter should be set to NULL.

FVID2 Queue

This API is used to submit a video buffer to the driver for display operation. This is a non-blocking call and could be called from task or ISR context. Once the buffer is queued the application loses ownership of the buffer and is not suppose to modify or use the buffer. When driver is openned under `VPS_GRPX_NON_FRAME_BUFFER_MODE`, the buffers submitted by multiple API calls will be queued by the driver. While if driver is openned under `VPS_GRPX_FRAME_BUFFER_MODE`, only the buffer submitted by the last API call will be used by the driver.

```
Int32 FVID2_queue(FVID2_Handle handle,
            FVID2_FrameList *frameList,
            UInt32 streamId);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

frameList - Pointer to `FVID2_FrameList` structure containing the pointer to the FVID2 frames(Graphics Region). This parameter should not be NULL. In normal display operation, the number of frames passed using this call is one. When multiple window(region) configuration is set, this frame list contains the frames(regions) of all the multiple window(region) buffers representing a single composited graphics frame.

streamId - Not used currently. This parameter should be set to 0.

FVID2 Dequeue

This API is used by the application to get ownership of a displayed video buffer from the display driver. This is a non-blocking call and could be called from task or ISR context. If the driver is opened under `VPS_GRPX_FRAME_BUFFER_MODE`, this API is not available and a error is returned to caller.

```
Int32 FVID2_dequeue(FVID2_Handle handle,
            FVID2_FrameList *frameList,
            UInt32 streamId,
            UInt32 timeout);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

frameList - Pointer to `FVID2_FrameList` structure where the driver will copy the displayed FVID2 frames. This parameter should not be NULL. In normal display operation, the number of frames(regions) passed using this call is one. When multiple window(region) configuration is set, this frame list contains the frames(regions) of all the multiple window(region) buffers representing a single composited graphics frame. For queuing multiple window(region) buffers, the frames(regions) should be given from top to bottom sequence.

streamId - Not used currently. This parameter should be set to 0.

timeout - Not used currently as only non-blocking queue/dequeue operation is supported. This parameter should be set to FVID2_TIMEOUT_NONE.

Delete Phase

In this phase FVID2 delete API is called to close the driver instance. All the resources are freed. Make sure display is stopped using FVID2_stop() before deleting a display instance. Once the driver is instance is closed it can be opened again with new configuration.

FVID2 Delete

This API is used to close the display driver. This is a blocking call and returns after closing the handle.

Note
Closing the driver will implicitly stop the display if stop IOCTl is not called by the application.

```
Int32 FVID2_delete(FVID2_Handle handle, Ptr deleteArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

deleteArgs - Not used currently. This parameter should be set to NULL.

System De-Init Phase

FVID2 de-Init

Driver sub-system gets de-initialized as a part of this phase. Here all resources acquired during system initialization are free'ed. Make sure all driver instances deleted before calling this API. Display sub-system de-init happens as part of overall FVID2 system de-init. Typically this is done during system shutdown.

```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

Sample Application

Single Region Display through HDMI VENC

This example illustrates the feature of graphics Path 0 by displaying one 240 x 160 RGB888 image on OFF-CHIP HDMI Encoder through DVO2 VENC configured with 1080P 30 FPS. This also illustrates the FVID2 queue/dequeue usage from Task context.

- Please refer Common Steps for connecting ccs to TI816x, running gel file etc.
- Load *hdvpss_examples_grpxDisplay.xem3* executable file found at *$(rel_folder)\build\example-name\bin\ti816x-evm\example-name-whole-program-debug.xem3*

to DSS M3 debug session

- Run the application and it will automatically load color bar frames.

```
Loading 3 graphics frames of size 240x160 to location: @BE000000 ... Image loading done ...
```

- This will display single moving graphics region in the video probe with runtime on/off stenciling and scaling features. Those changes were present every 100 frames, which is controll by the RTS_SWITCH_RATE.
- Application will stop after displaying 3000 frames
- Configuration Options: To display in graphics path 1/2, change the macro `DRIVER_INSTANCE` to `VPS_DISP_INST_GRPX1/2`
- Configuration Options: To change the number of frames to display, change the macro `TOTAL_LOOP_COUNT` to any desired value greater than 2
- Configuration Options: Change RT_SWITCH_RATE to adjust how ofter the runtime change is performed. To disable this function, set the RT_SWITCH_RATE bigger than the total loop count.

Multiple Regions Display through HDMI VENC

This example illustrates the feature of graphics Path 0 by displaying two 240 x 160 RGB888 images on OFF-CHIP HDMI encoder through DVO2 VENC configured for 1080P 30 FPS. This also illustrates the FVID2 queue/dequeue usage from Task context.

- Please refer Common Steps for connecting ccs to TI816x, running gel file etc.
- Load *hdvpss_examples_grpxDisplayMultiReg.xem3* executable file found at *$(rel_folder)\build\example-name\bin\ti816x-evm\example-name-whole-program-debug.xem3*

to DSS M3 debug session

- Run the application and it will automatically load the color bar test image.

```
Loading 6 graphics frames of size 240x160 to location: @BE000000 ... Image loading done ...
```

- This will display two moving graphics regions in the video probe with runtime enable/disable stenciling and scaling ratio. Those changes were present every 100 frames, which is controll by the RTS_SWITCH_RATE.
- Application will stop after displaying 3000 frames
- Configuration Options: To display in graphics path 1/2, change the macro `DRIVER_INSTANCE` to `VPS_DISP_INST_GRPX1/2`
- Configuration Options: To change the number of frames to display, change the macro `TOTAL_LOOP_COUNT` to any desired value greater than 2
- Configuration Options: Change RT_SWITCH_RATE to adjust how ofter the runtime change is performed. To disable this function, set the RT_SWITCH_RATE bigger than the total loop count.

UserGuideHdvpssM2mDriver

Memory to Memory Drivers

Introduction

Memory to Memory drivers takes the video buffer from the memory, optionally process the buffer, processing done on the buffer depends on the specific memory to memory driver and puts it back to memory. Memory to memory driver follows the FVID2 interface for the applications.

Following are the general feature set for the memory to memory drivers:

- All the memory to memory driver supports multiple handle. This means the driver can be opened multiple times.
- All the memory drivers supports multiple channels request submission per handle. Multiple channels means the video stream coming from multiple streams like frames coming from decoder over network, multiple capture streams each having different frame parameters like height, width etc.
- Memory driver supports parameter configuration for the buffer processing per handle or per channel of the handle. There can be individual set of parameters for each channel of the handle like height, width, data format etc or else application can have the same parameters for all the channels of the handle.
- Application can submit multiple channels for processing in a single request call
- FVID2_processFrames (queue) and FVID2_getProcessFrames (de-queue) FVID2 calls for all the memory to memory drivers are non blocking. While the control commands like programming of the scalar coefficients are blocking.
- All memory to memory driver calls the application call back function on completion of the request. Application should de-queue the request after the callback.

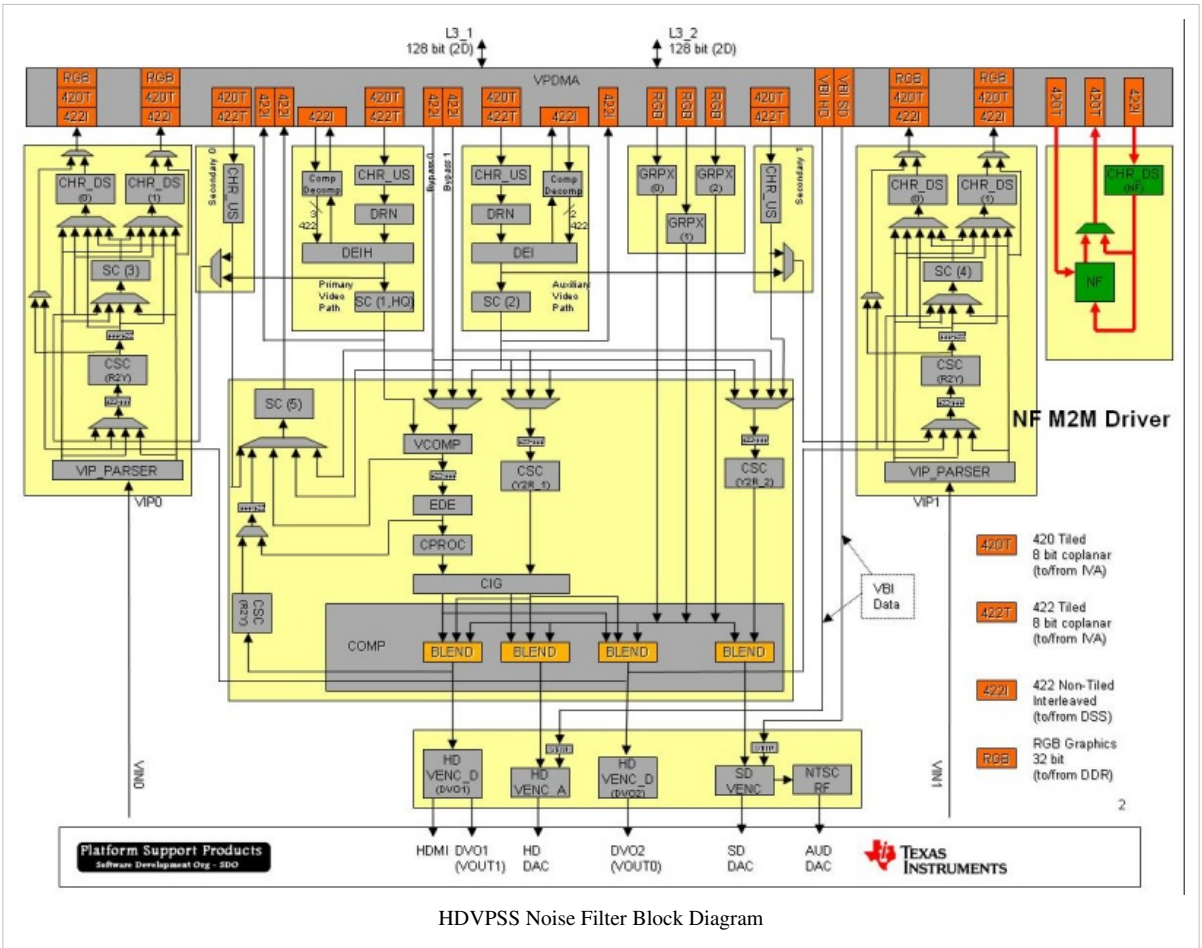
Noise Filter (NSF) - Memory to Memory Driver

Introduction

Noise filter (NF or NSF) driver allows user to filter noise from video data by processing them through the noise filter hardware. The NF hardware supports spatial as well as temporal noise filtering.

When temporal noise filtering is enabled, the hardware needs the previous noise filtered output as one of the inputs. When temporal noise filter is disabled (`VPS_NSF_BYPASS_MODE_SNF_TNF`), this previous noise filtered frame is not required. It is possible to bypass both spatial as well as temporal noise filter (`VPS_NSF_BYPASS_MODE_SNF_TNF`), i.e. driver can be used for only YUV422 to YUV420 chroma downsampling. In this case too, previous noise filtered frame is not required.

The data paths supported by the current driver implementation are shown in the figure below:



HDVPSS Noise Filter Block Diagram

The features and limitations of current driver implementation are listed in subsequent sections.

Important

The features supported or NOT supported in any release of the NSF driver may vary from one HDVPSS driver release to another. See respective release notes for exact release specific details.

Features Supported

- Input formats:
  - YUV422, non-tiled memory, YUYV interleaved data - this is the only input format supported by the NSF hardware
- Output formats:
  - YUV420T, tiled memory, YUV420 semi-planer data - this is the only output format supported by the NSF hardware
- Multi-channel support - upto `VPS_NSF_MAX_CH_PER_HANDLE` channels per handle
- Multi-handle support - upto `VPS_NSF_MAX_HANDLES` handles per system
- Configurable input size (width, height, startX, startY, pitch) per channel
- Configurable output pitch per channel. Output size is always same as input size
- Configurable noise filter processing parameters like filter strength, filter threshold per channel
- Configurable noise filter operation mode per channel like temporal NF bypass, spatial NF bypass, all NF bypass i.e. only chroma downsample
- Run-time input size and noise filter parameter change via FVID2 control API
- Non-blocking FVID2 process frames and get processed frames API support

- Tiler support for YUV420 output, YUV420 previous filtered input
- Slice based NF (Will be supported only on NETRA)

Features Not Supported

- Slice based NF (Centarus)

Software Application Interfaces

The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: Here the driver is used to filter or process the frames continuously
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is deinitialized

The subsequent sections describe each phase in detail.

System Init Phase

The NSF driver sub-system initialization happens as part of overall HDVPSS system init. Below code shows the FVID2 API used to initialize the overall HDVPSS subsystem. This API must be the the first API call before making any other FVID2 calls.

FVID2 Init

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

```
#include "ti/psp/vps/vps_m2mNsf.h"
Int32 mySysInit() {
      Int32 status;
      status = FVID2_init(NULL);
      if(status!=FVID2_SOK) {
            // error in HDVPSS driver system initialization
      }
      return status;
}
```

Internally, following happens when NSF driver initialization is done:

- Hardware resources like interrupts, hardware lists are allocated
- NF Hardware is reset to known state and NF related muxes are configured
- Driver name is registered with FVID2 sub-system

Create Phase

In this phase, user application opens or creates a driver instance. Upto `VPS_NSF_MAX_HANDLES` (defined in `vps_m2mNsf.h`) driver instances can be opened by a user. Upto `VPS_NSF_MAX_CH_PER_HANDLE` (defined in `vps_m2mNsf.h`) channels can be associated with a given handle. Upto `VPS_NSF_MAX_CH_IN_ALL_HANDLES` (defined in `vps_m2mNsf.h`) channels can be associated when all handles are considered together. When any of the max handles, max channels per handles or max channels in all handles limit is exceeded by user, error is returned during FVID2 create.

User can pass a number of parameters during create which controls the mode in which the driver instance gets created. For e.g., number of channels, width and height of each channel, NF processing parameters for each channel. Refer to `M2M Noise Filter API` section in HDVPSS API Guide for detailed list of create time parameters.

FVID2 create

```
FVID2_Handle FVID2_create(UInt32 drvId,
                  UInt32 instanceId,
                  Ptr createArgs,
                  Ptr createStatusArgs,
                  const FVID2_CbParams *cbParams);
```

This API is used to open the driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

drvId - Pass `FVID2_VPS_M2M_NSF_DRV` to open noise filter memory driver.

instanceId - Pass `VPS_M2M_INST_NF0` macro to open the only instance of noise filter driver

createArgs - Pass a pointer to `Vps_NsfCreateParams` structure containing the valid parameters. This parameter should not be null

createStatusArgs - Pass a pointer to `Vps_NsfCreateStatus` structure. Driver returns status in this structure. This parameter can be NULL.

cbParams - Pass the pointer to `FVID2_DrvCbParams`. These call back parameters are used to indicate the successful processing of the frame or the error frames.

As an example, FVID2 create call, used to create a NF driver instance, is shown below:

```
#include "ti/psp/vps/vps_m2mNsf.h"

FVID2_Handle  fvidHandle;
Vps_NsfCreateParams createArgs;
Vps_NsfCreateStatus createStatus;
FVID2_CbParams  cbPrm;
UInt16 chId;
Vps_NsfDataFormat dataFormat[VPS_NSF_MAX_CH_PER_HANDLE];
Vps_NsfProcessingParams processingCfg[VPS_NSF_MAX_CH_PER_HANDLE];
Vps_NsfDataFormat *pDataFormat;
Vps_NsfProcessingParams *pProcessingParams;

createArgs.numCh = VPS_NSF_MAX_CH_PER_HANDLE;
createArgs.dataFormat = dataFormat;
createArgs.processingCfg = processingCfg;
cbPrm.cbFxn = myCallbackFunc;
cbPrm.appData = NULL;
cbPrm.errCbFxn = NULL;
cbPrm.errData = NULL;

for(chId=0; chId < createArgs.numCh; ch++)
{
      pDataFormat =  & createArgs.dataFormat[chId];
      pProcessingParams =  & createArgs.processingCfg[chId];
```

```
      pDataFormat->channelNum = chId;
      pDataFormat->inMemType = VPS_VPDMA_MT_NONTILEDMEM;
      pDataFormat->outMemType = VPS_VPDMA_MT_NONTILEDMEM;
      pDataFormat->inDataFormat = FVID2_DF_YUV422I_YUYV;
      pDataFormat->inFrameWidth = 720;
      pDataFormat->inFrameHeight = 480;
      pDataFormat->inCropCfg.cropStartX = 0;
      pDataFormat->inCropCfg.cropStartY = 0;
      pDataFormat->inCropCfg.cropWidth = pDataFormat- >
inFrameWidth;
      pDataFormat->inCropCfg.cropHeight = pDataFormat- >
inFrameHeight;
      pDataFormat->inPitch = ALIGN(pDataFormat- > inFrameWidth,
32)*2;
      pDataFormat->outDataFormat = FVID2_DF_YUV420SP_UV;
      pDataFormat->outPitch[0] = pDataFormat- > inPitch/2;
      pDataFormat->outPitch[1] = pDataFormat- > outPitch[0];
      pProcessingParams->channelNum  = chId;
      pProcessingParams->bypassMode  = VPS_NSF_BYPASS_MODE_NONE;
      pProcessingParams->enableFrameNoiseAutoCalc = TRUE;
      pProcessingParams->resetFrameNoiseCalc = TRUE;
      pProcessingParams->enableSliceMode = FALSE;
      pProcessingParams->numLinesPerSlice = 128;
      pProcessingParams->staticFrameNoise[0]     = 0;
      pProcessingParams->staticFrameNoise[1]     = 0;
      pProcessingParams->staticFrameNoise[2]     = 0;
      pProcessingParams->spatialStrengthLow[0]
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->spatialStrengthLow[1]
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->spatialStrengthLow[2]
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->spatialStrengthHigh[0]
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->spatialStrengthHigh[1]
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->spatialStrengthHigh[2]
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->temporalStrength
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->temporalTriggerNoise
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->noiseIirCoeff
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->maxNoise
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->pureBlackThres
```

```
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
      pProcessingParams->pureWhiteThres
      = VPS_NSF_PROCESSING_PARAMS_DEFAULT;
}

fvidHandle = FVID2_create(
                  FVID2_VPS_M2M_NSF_DRV,
                  VPS_M2M_INST_NF0,
                  & createArgs,
                  & createStatus,
                  & cbPrm
                  );
if(fvidHandle==NULL)
{
      // error in FVID2 handle creation
}
```

Internally, following happens when FVID2 create is called:

- Software resources like semaphores, queues are allocated, depending on the create parameters that are passed
- NO hardware register or VPDMA descriptor programming takes place during create.

Run Phase

In this phase the driver can be used to process frames continuously (`FVID2_processFrames()`). Once the frames are noise filtered, the driver generates a callback to indicate completion of frame processing, at which point user could get the processed frames from the driver (`FVID2_getProcessedFrames()`).

Like other memory-to-memory drivers, user can send multiple frames from different channels for processing in one request via `FVID2_processFrames()` function call. Also, multiple such requests can be queued inside driver without user having to wait for completion of a previously submitted request.

FVID2_processFrames

```
Int32 FVID2_processFrames(FVID2_Handle handle,
                  FVID2_ProcessList *processList);
```

This is used to submit the frames for processing. Processlist contains the frames from various channels to be processed. Application can open the driver for N channels and it can submit request for the M channels where M <= N. Processlist is returned to the driver and it can use the processlist for submitting the next request whereas all the elements inside the processlist is driver's ownership and can be reused by application only after de-queuing the request. This is a non blocking call and requests are queued inside the driver for processing. Driver calls the application call back once the hardware completes processing the request.

handle - Pass the handle returned to the application while opening of the driver.

processList - Pass the processlist containing the frames to be processed. User can also pass the run time parameters with the processlist to change the parameters run time. Run time parameters only needs to be passed once. Subsequent process_frames function will be operated with the last run time parameters passed. If user wants to updated that again new set of run time parameters needs to be passed. Noise Filter driver supports run time parameters change in synchronous with the submitted frame. Run time parameter structure supported by Noise Filter driver is `Vps_M2mNsfRtParams`.

Queuing multiple requests

`FVID2_processFrames()` can be called multiple number of times to queue multiple requests to the driver. Once the driver internal request queue is full, no further requests will be accepted by the driver and error is returned. The number of requests that can be queued without any error or blocking is made known to the user via status, returned during create phase (`Vps_NsfCreateStatus.maxReqInQueue`).

Queuing multiple frames from same channel

`FVID2_processFrames()` can be used to queue frames from the same channel either in the same API call or multiple API calls. However, since the internal channel specific memory (queue) is statically allocated, the number of frames per channel that can be queued is limited to `Vps_NsfCreateStatus.maxFramesPerChInQueue`. Once the per channel queue becomes full, no further frames will be accepted by the driver and error will be returned. Note, when, for a channel, frame submission error is returned, the current request is aborted for all channels in that request.

Maximum frames in one request

The number of frames that be submitted in a request is also limited to `Vps_NsfCreateParams.maxFramesInProcessFrames`. This will be at least equal to the number of channels in the handle (`Vps_NsfCreateParams.numCh`)

Below example shows the APIs that are used to submit frames to the driver for processing. Once the submitted frames are processed by NF driver, it calls a user specified callback to indicate that the frames have been processed.

```
#include "ti/psp/vps/vps_m2mNsf.h"
/* NSF driver request information */
typedef struct
{
      /* Input frame pointers */
      FVID2_Frame *inFrames[VPS_NSF_MAX_CH_PER_HANDLE];
      /* Previous output frame pointers */
      FVID2_Frame *prevOutFrames[VPS_NSF_MAX_CH_PER_HANDLE];
      /* Output frame pointers */
      FVID2_Frame *outFrames[VPS_NSF_MAX_CH_PER_HANDLE];
      /* Input frame list
      0 - for input frames
      1 - for previous output frames
      */
      FVID2_FrameList inFrameList[2];
      /* Output frame list
      0 - for output frames
      */
      FVID2_FrameList outFrameList[1];
      /* Process list thats submitted to driver in this request */
      FVID2_ProcessList processList;
} NsfApp_ReqObj;

/* make process list  */
Int32 NsfApp_drvObjMakeReq(NsfApp_ReqObj *reqObj)
{
      UInt32 chId;
      Int32 prevOutFrameId, numFramesInReq;
```

```
      /* A request consists of frame from every channel.
      Note, this is just the way test case is written, as such a
reqeust can
      mix of requests for same or different channels and that too in
any order
      */

      numFramesInReq = createArgs.numCh;
      reqObj->inFrameList[0].frames = reqObj->inFrames;
      reqObj->inFrameList[0].numFrames = numFramesInReq;
      reqObj->inFrameList[0].perListCfg = NULL;
      reqObj->inFrameList[0].reserved = NULL;

      /* init inFrameList [1] */
      reqObj->inFrameList[1].frames = reqObj->prevOutFrames;
      reqObj->inFrameList[1].numFrames = numFramesInReq;
      reqObj->inFrameList[1].perListCfg = NULL;
      reqObj->inFrameList[1].reserved = NULL;

      /* init outFrameList [0] */
      reqObj->outFrameList[0].frames = reqObj->outFrames;
      reqObj->outFrameList[0].numFrames = numFramesInReq;
      reqObj->outFrameList[0].perListCfg = NULL;
      reqObj->outFrameList[0].reserved = NULL;

      /* init processList */
      reqObj->processList.inFrameList[0] = &
reqObj->inFrameList[0];
      reqObj->processList.inFrameList[1] =  &
reqObj->inFrameList[1];
      reqObj->processList.outFrameList[0] =  &
reqObj->outFrameList[0];
      reqObj->processList.numInLists = 2;
      reqObj->processList.numOutLists = 1;
      reqObj->processList.reserved = NULL;

      /* for each channel in request obj do ... */
      for(chId=0; chId < numFramesInReq; chId++)
      {
            /*  Set input and output frame pointers */
            reqObj->inFrames[chId]      = get input frame pointer;
            reqObj->prevOutFrames[chId] = get previous output frame
pointer;
            reqObj->outFrames[chId]     = get output frame pointer;
      }
      return FVID2_SOK;
}
```

```
/*  Use driver to process frames*/
Int32 NsfApp_drvObjProcessFrames()
{
      NsfApp_ReqObj pReqObj;
      Int32 status, reqId, numReqInQueue;

      /*
      get max request to queue, based on driver status thats
      returned during create
      */

      numReqInQueue = createStatus.maxReqInQueue;

      /* for each request that is to be submitted ... */
      for(reqId=0; reqId < numReqInQueue; reqId++)
      {
            /* get current free request object  */
            pReqObj =  & reqObj[reqId];

            /* make the request object  */
            NsfApp_drvObjMakeReq(pReqObj);

            /* Submit  request to driver  */
            status = FVID2_processFrames(fvidHandle,  &
pReqObj->processList);

            if(status!=FVID2_SOK)
            {
                  /* Error in request submission  */
            }
      }
      return status;
}
```

FVID2_getProcessedFrames

```
Int32 FVID2_getProcessedFrames(FVID2_Handle handle,
                        FVID2_ProcessList *processList,
                        UInt32 timeout);
```

This is used to de-queue the already processed request. This is again a non blocking call and if there are requests in the driver to be de-queued it will return the dequeued frames in the processlist else it will return with error.

handle - Pass the handle returned to the application while opening of the driver.

processList - Pass the pointer to the processlist. Driver will copy the pointer to the processed framelist to the processlist.

timeout - Currently unused arguments. Driver ignores this.

When the user callback gets called, user can get the processed frames from the NF driver. The processed frames can be retrieved from the driver in the callback itself or later in some application task context.

An example is shown below:

```
FVID2_ProcessList processList;

Int32 timeout;

timeout = BIOS_NO_WAIT; // non-blocking

/* get processed frames from driver */
status = FVID2_getProcessedFrames(fvidHandle,  & processList,
timeout);
if(status!=FVID2_SOK)
{
      /* Error in getting processed frames  */
}
else
{
      /* success  */
}
```

Run time parameter change

There are two ways of changing the runtime parameters.

IOCTL Method

Some parameters like input size, noise filter processing can be changed while the driver is in running phase. Once a parameter is changed, the effect of the change will happen only for the subsequent frames that are submitted. For current frames, new parameters will not take effect. These parameters can be changed independently for each channel.

FVID2 IOCTLs, as shown below, can be used for run-time parameter change:

- `IOCTL_VPS_NSF_SET_PROCESSING_CFG` - this can be used to change noise filter processing parameters like operation mode (TNF/SNF ON/OFF), filter strengths, filter threshold, reset filter state etc.
- `IOCTL_VPS_NSF_SET_DATA_FORMAT` - this can be used to change input size, input pitch, input area selection (crop) and output pitch.

An example is shown below:

```
#include "ti/psp/vps/vps_m2mNsf.h"
Int32 NsfApp_drvObjUpdateParams()
{
      Int32 status;
      UInt32 chId;
      Vps_NsfDataFormat *nsfDataFormat;

      /* for all channels */
      for(chId=0; chId < createArgs.numCh; chId++)
      {
            /* channel data format  */
```

```
            nsfDataFormat = &createArgs.dataFormat[chId];

            /* half the input width x height for the channel */
            nsfDataFormat->inMemType = VPS_VPDMA_MT_NONTILEDMEM;
            nsfDataFormat->outMemType = VPS_VPDMA_MT_NONTILEDMEM;
            nsfDataFormat->inFrameWidth   /= 2;
            nsfDataFormat->inFrameHeight  /= 2;
            nsfDataFormat->inCropCfg.cropStartX       = 0;
            nsfDataFormat->inCropCfg.cropStartY       = 0;
            nsfDataFormat->inCropCfg.cropWidth        =
nsfDataFormat->inFrameWidth;
            nsfDataFormat->inCropCfg.cropHeight       =
nsfDataFormat->inFrameHeight;

            /* rest of the parameters are kept same as create time
parameters */
            /* Do IOCTL to change the new channel information
            This updated information will get reflected from
            next submission to the driver
            Pending submission will still use old channel information
            */
            status = FVID2_control(fvidHandle,
                                   IOCTL_VPS_NSF_SET_DATA_FORMAT,
                                   nsfDataFormat,
                                   NULL
                                   );
            assert(status==FVID2_SOK);
      }
      return 0;
}
```

Runtime parameters passed with the frame

User can also pass the run time parameters with the processlist to change the parameters run time. Run time parameters only needs to be passed once. Subsequent process_frames function will be operated with the last run time parameters passed. If user wants to updated that again new set of run time parameters needs to be passed. Noise Filter driver supports run time parameters change in synchronous with the submitted frame. Run time parameter structure supported by Noise Filter driver is Vps_M2mNsfRtParams.

Delete Phase

In this phase, FVID2 delete API is called to free all resources allocated during create. Make sure no frames are submitted/queued to the driver when this API is called.

The FVID2 delete API call is shown below:

```
#include "ti/psp/vps/vps_m2mNsf.h"

FVID2_delete(fvidHandle, NULL);
```

System De-init Phase

In this phase, NF sub-system is de-initialized. Here, all resources acquired during system initialization are freed. Make sure all NF handles are deleted before calling this API. NF sub-system de-init happens as part of overall FVID2 system de-init. Typically this is done during system shutdown.

```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

```
#include "ti/psp/vps/vps_capture.h"

Void mySysDeInit() {
      FVID2_deInit(NULL);
}
```

Sample Application

This section shows how to run the sample application for NF driver. The sample application source code is located at the below path: `\packages\ti\psp\examples\common\vps\m2m\m2mNsf`

Running the sample application

The Memory-to-memory Noise Filter application executes the NF driver in many different modes, like single handle, single channel, multi handle, multi channel, with callback, without callback and so on. The sample code does some limited data verification check to make sure that output data is fine. It also prints performance information including frame per second achieved and the CPU load.

The application takes the input buffer for processing, optionally processes the buffers by running through the list of pre-configured options one by one, and then writes the buffer(s) back to memory.

Following are the steps to run the application:

- Please refer Common Steps for connecting CCS to TI816x, running gel file etc.
- Load *hdvpss_examples_m2mNsf_m3vpss_whole_program_debug.xem3*' at *$(rel_folder)\build\example-name\bin\ti816x-evm\example-name_whole_program_debug.xem3*

to DSS M3 debug session

- Run the application.
- The application will halt for the user to load the input frames. Using `loadRaw` command load images on the memory location mentioned in the console print.

The command to be used for loading the image into the memory buffer shall be printed on the console. For example, the command for loading the images is similar to the below:

```
loadRaw(<addr>, 0,
"<filePath>\\<fileName>_yuyv422_prog_packed_640_480.tigf",
32, false);
```

- Press any key after loading
- The program continues running, finishes processing and then waits for the user to save the processed image.
- Save the processed image(s). Using `saveRaw` command, the image file can be saved. The command to be used for saving the image file from the memory buffer shall be printed on the console. For example, the command for saving the image file is similar to the below:

```
saveRaw(0, <addr>,
"<filePath>\\<fileName>_nv12_prog_packed_640_480.tigf",
1152000, 32, true);
```

Warning

If the processor is not halted or waiting for console input when saving the images, the images will be seen green as the data dumped will be 0x00.

- View the saved image using any external YUV image viewer.
- The program runs again for the next option in the list till it waits for the user to save the processed image. This continues till all the options are completed.

Once application execution is complete, to re-run the application, just reset the CPU, reload and run as before.

Sample output printed on the CCS console is shown below:

```
  NSFAPP: Load YUYV422 test data ( 640 x 480, 10 frames ) @ 0xa0800000
!!!
  loadRaw( 0xa0800000, 0, "<my
folder>\noisyVideo_yuyv422_prog_packed_640_480.tigf", 32, false);
  NSFAPP: Press Any Key to Continue ... !!!

  NSFAPP: NsfApp_init() - DONE !!!
  NSFAPP: HANDLES 1: CHANNELS 1 : RUN COUNT 10: UPDATE_PRM_RT 0: MODE:
0 !!!
 M2mNsf:Output Buffers Start Address 0xa0ddc000
 M2mNsf:Save output file with command:saveRaw(0, 0xa0ddc000,
"C:\\0nsfWbOut_nv12_prog_packed_640_480.tigf", 1152000, 32, true);
  NSFAPP: 0: NsfApp_initDrvObj() - DONE !!!
  NSFAPP:     5.2 s: Frames = 5224 (1044 fps)
  NSFAPP: 0: NsfApp_deInitDrvObj() - DONE !!!
  NSFAPP: HANDLES 1: CHANNELS 1 : RUN COUNT 10: UPDATE_PRM_RT 0!!! -
DONE


  Execution       Total Total   Total     Total D1@60Hz  CPU
       Time      Frames   FPS Mpixels Mpixels/s    Ch's Load
  ========================================================
     10.1 s   10436  1043    1803       180      17    9


  10250: LOAD: CPU: 8 HWI: 3, SWI:0

  10251: PRF : NSFAPP: : t: 10000 ms, c: 1, f: 2609, fps: 260, fpc:
2609
  NSFAPP: Press Any Key to continue after saving output image ... !!!

  NSFAPP: HANDLES 1: CHANNELS 1 : RUN COUNT 10: UPDATE_PRM_RT 0: MODE:
3 !!!
 M2mNsf:Output Buffers Start Address 0xa0ddc000
 M2mNsf:Save output file with command:saveRaw(0, 0xa0ddc000,
```

```
"C:\\1nsfWbOut_nv12_prog_packed_640_480.tigf", 1152000, 32, true);
  NSFAPP: 0: NsfApp_initDrvObj() - DONE !!!
  NSFAPP:     5.1 s: Frames = 5248 (1049 fps)
  NSFAPP: 0: NsfApp_deInitDrvObj() - DONE !!!
  NSFAPP: HANDLES 1: CHANNELS 1 : RUN COUNT 10: UPDATE_PRM_RT 0!!! -
DONE


  Execution      Total Total   Total     Total D1@60Hz  CPU
       Time     Frames   FPS Mpixels Mpixels/s    Ch's Load
  ============================================================
     10.0 s   10484  1048    1811       181      17    9


  20259: LOAD: CPU: 8 HWI: 3, SWI:0
```

SubFrame Based Noise Filtering

Introduction

Applications which requires low latency in video processing, like Video communications, divides video frame in to multiple parts, called subframes and do all processing at subframe level. This results in total end2end delay reduction and thus enhances user experience.

This chapter describes support of subframe based Noise Filtering in memory to memory drivers, application software interfaces, and sample application usage.

Software Overview

Application can enable subframe processing for each channel at Create time by setting enable flag and providing number of lines per subframe in Channel create parameters.

For Processing, Application should pass subframe number and number of lines per subframe as input along with the frame start address, for each subframe. Driver will process this subframe and updates the number of lines available in output frame for further processing by other modules.

Software Application Interfaces

This section describes SubFrame processing related structures and parameters setting in FVID2 level functions.

Application Interfaces

FVID2_create

```
FVID2_Handle FVID2_create(UInt32 drvId,
                  UInt32 instanceId,
                  Ptr createArgs,
                  Ptr createStatusArgs,
                  const FVID2_CbParams *cbParams);
```

This API is used to open the driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

To enable subframe processing, `Vps_SubFrameParams` structure elements inside `createArgs` structure of corresponding driver needs to be populated with below values:

- `Vps_SubFrameParams->subFrameModeEnable` should be set to TRUE
- `Vps_SubFrameParams->numLinesPerSubFrame` should be set to a value equal to NumberOfLinesPerSubFrame.
- NumberOfLinesPerSubFrame should always be multiple of 32 as NF internally processes on a 32x32 block

FVID2_processFrames

```
Int32 FVID2_processFrames(FVID2_Handle handle,
                  FVID2_ProcessList *processList);
```

This is used to submit the frames for processing. Processlist contains the frames from various channels to be processed. Application can open the driver for N channels and it can submit request for the M channels where M <= N. **Processlist is returned to the driver and** it can use the processlist for submitting the next request whereas all the elements inside the processlist is driver's ownership and can be reused by application only after de-queuing the request. This is a non blocking call and requests are queued inside the driver for processing. Driver calls the application call back once the hardware completes processing the request.

handle - Pass the handle returned to the application while opening of the driver.

processList - Pass the processlist containing the frames to be processed.

For channels where subframe processing is enabled, `FVID2_SubFrameInfo` structure (inside `FVID2_Frame`) of the frames related to this channel needs to be populated with below values.

- `FVID2_SubFrameInfo->subFrameNum` should be set to '0' for first subframe and needs to be incremented by '1' for subsequent subframes.
- `FVID2_SubFrameInfo->numInLines` should be set to Number of lines per subframe.

Warning
SubFrame parameter interface may change in future as this interface is currently experimental.

FVID2_getProcessedFrames

```
Int32 FVID2_getProcessedFrames(FVID2_Handle handle,
                       FVID2_ProcessList *processList,
                       UInt32 timeout);
```

This is used to de-queue the already processed request. This is again a non blocking call and if there are requests in the driver to be de-queued it will return the dequeued frames in the processlist else it will return with error.

handle - Pass the handle returned to the application while opening of the driver.

processList - Pass the pointer to the processlist. Driver will copy the pointer to the processed framelist to the processlist.

For the channels where subframe processing is enabled, `FVID2_SubFrameInfo` structure of output frames will be updated by driver.

- `FVID2_SubFrameInfo->subFrameNum` contains latest subframe number processed by driver, 0 to N-1.
- `FVID2_SubFrameInfo->numOutLines` contains Number of lines avaialble in output frame after processing current subframe.

timeout - Currently unused arguments. Driver ignores this.

Sample Applications

This section shows how to run the sample application for slice based NF driver. The sample application source code is located at the below path: `\packages\ti\psp\examples\common\vps\m2m\m2mNsf_Subframe`

Running the sample application

To run the non-tiler memory sample application, load and run the `$(rel_folder)\build\example-name\bin\ti816x-evm\example-name-whole-program-debug.xem3` via CCS.

Once application execution is complete, to re-run the application, just reset the CPU, reload and run as before.

Some notes about the sample application:

- Please refer Common Steps for connecting CCS, running gel file etc.
- The sample application executes the NF driver in many different modes, and it takes 1920x1080 frame as input with slice size of 128.
- The sample code does some limited data verification check to make sure that output data is fine.

Sample output printed on the CCS console is shown below:

```
  NSFAPP: NsfApp_init() - DONE !!!
  NSFAPP: HANDLES 1: CHANNELS 1 : RUN COUNT 10: UPDATE_PRM_RT 0: MODE:
0 !!!
  NSFAPP: 0: NsfApp_initDrvObj() - DONE !!!
  NSFAPP:     5.2 s: Frames = 5224 (1044 fps)
  NSFAPP: 0: NsfApp_deInitDrvObj() - DONE !!!
  NSFAPP: HANDLES 1: CHANNELS 1 : RUN COUNT 10: UPDATE_PRM_RT 0!!! -
DONE


  Execution      Total Total   Total     Total D1@60Hz  CPU
       Time     Frames   FPS Mpixels Mpixels/s    Ch's Load
  ============================================================
     10.1 s   10436  1043    1803       180      17    9


  10250: LOAD: CPU: 8 HWI: 3, SWI:0

  10251: PRF : NSFAPP: : t: 10000 ms, c: 1, f: 2609, fps: 260, fpc:
2609
  NSFAPP: HANDLES 1: CHANNELS 1 : RUN COUNT 10: UPDATE_PRM_RT 0: MODE:
3 !!!
  NSFAPP: 0: NsfApp_initDrvObj() - DONE !!!
  NSFAPP:     5.1 s: Frames = 5248 (1049 fps)
  NSFAPP: 0: NsfApp_deInitDrvObj() - DONE !!!
  NSFAPP: HANDLES 1: CHANNELS 1 : RUN COUNT 10: UPDATE_PRM_RT 0!!! -
DONE


  Execution      Total Total   Total     Total D1@60Hz  CPU
```

```
      Time      Frames    FPS Mpixels Mpixels/s     Ch's Load
=============================================================
    10.0 s    10484   1048    1811        181       17     9


20259: LOAD: CPU: 8 HWI: 3, SWI:0
```

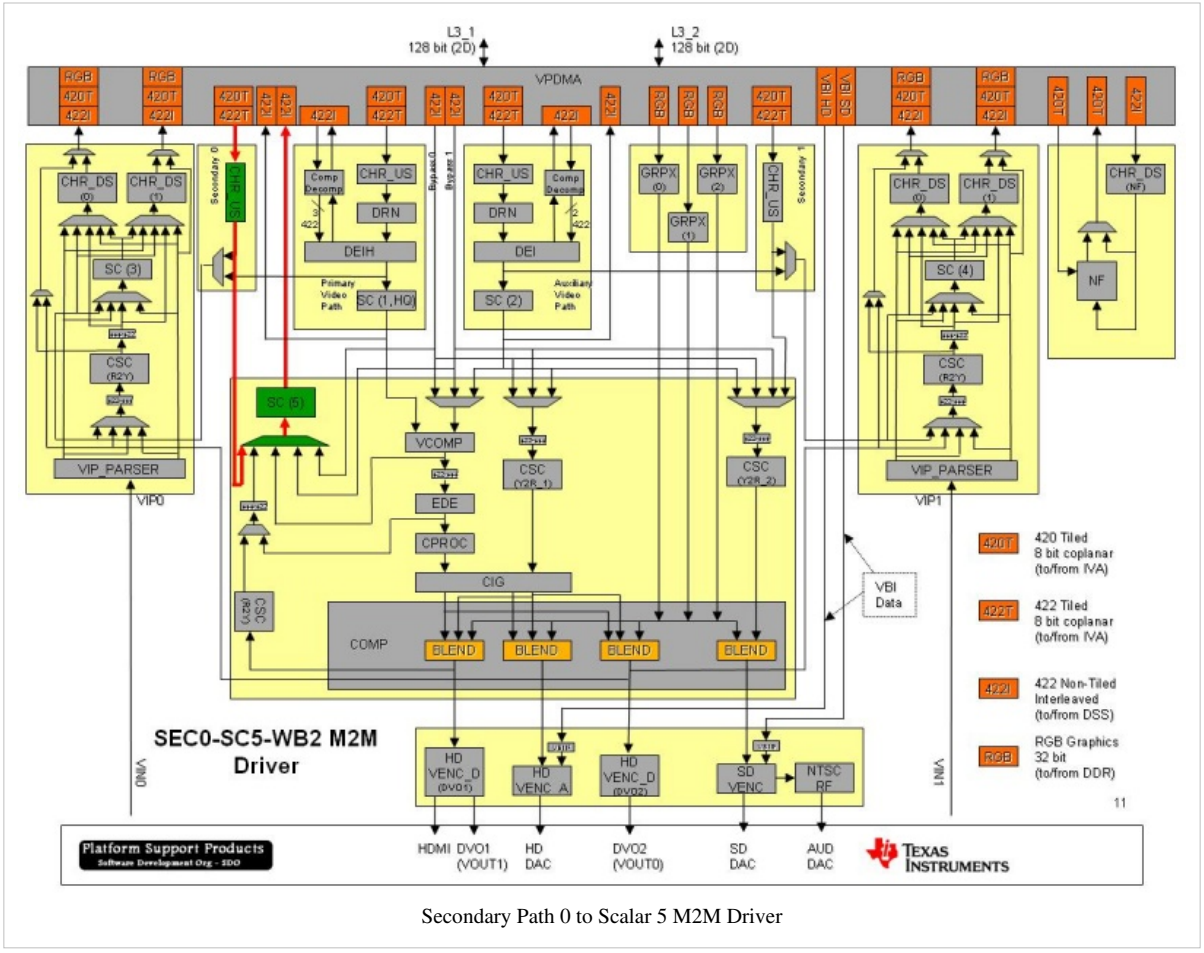
Scalar (SC) - Memory to Memory Driver

Introduction

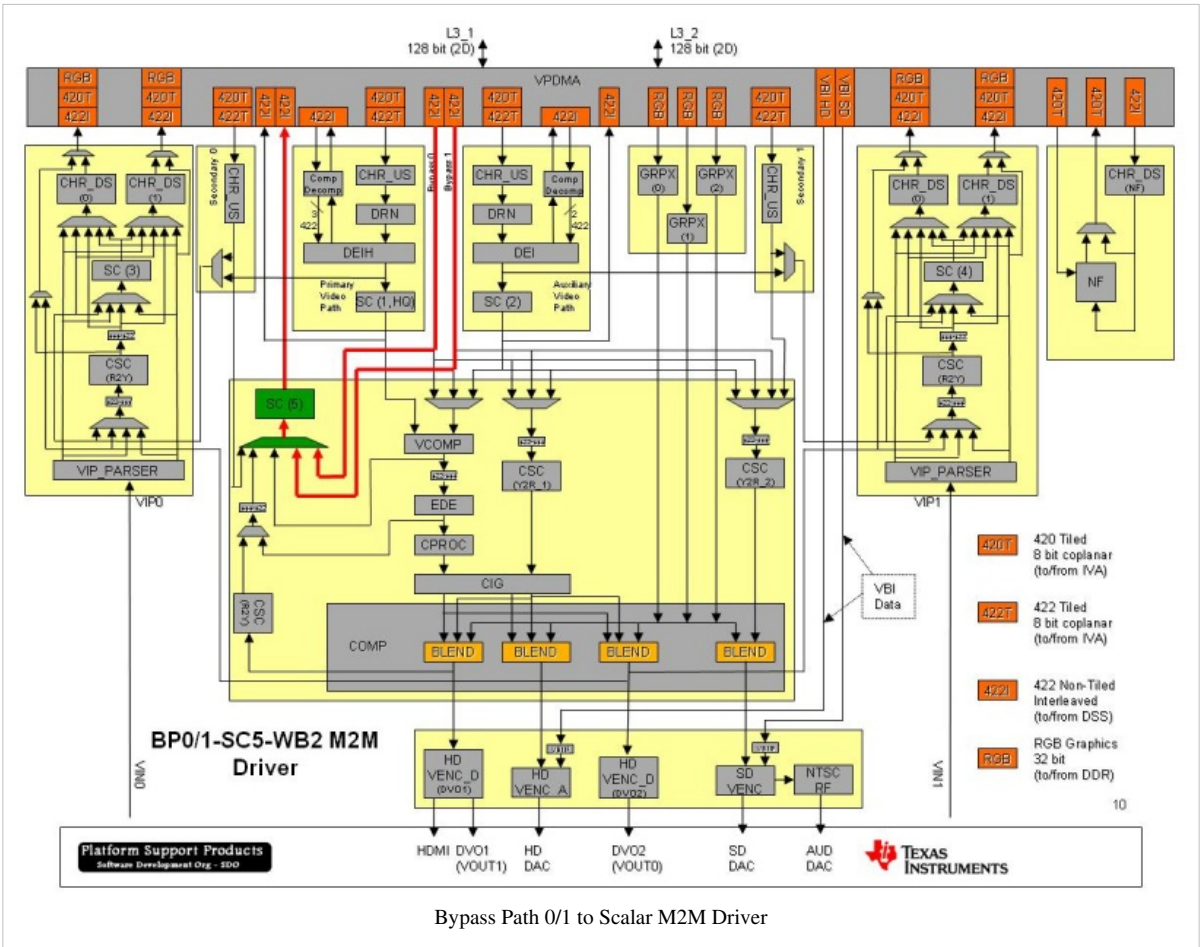
This chapter describes the hardware overview, application software interfaces, typical application flow and sample application usage.

Hardware Overview

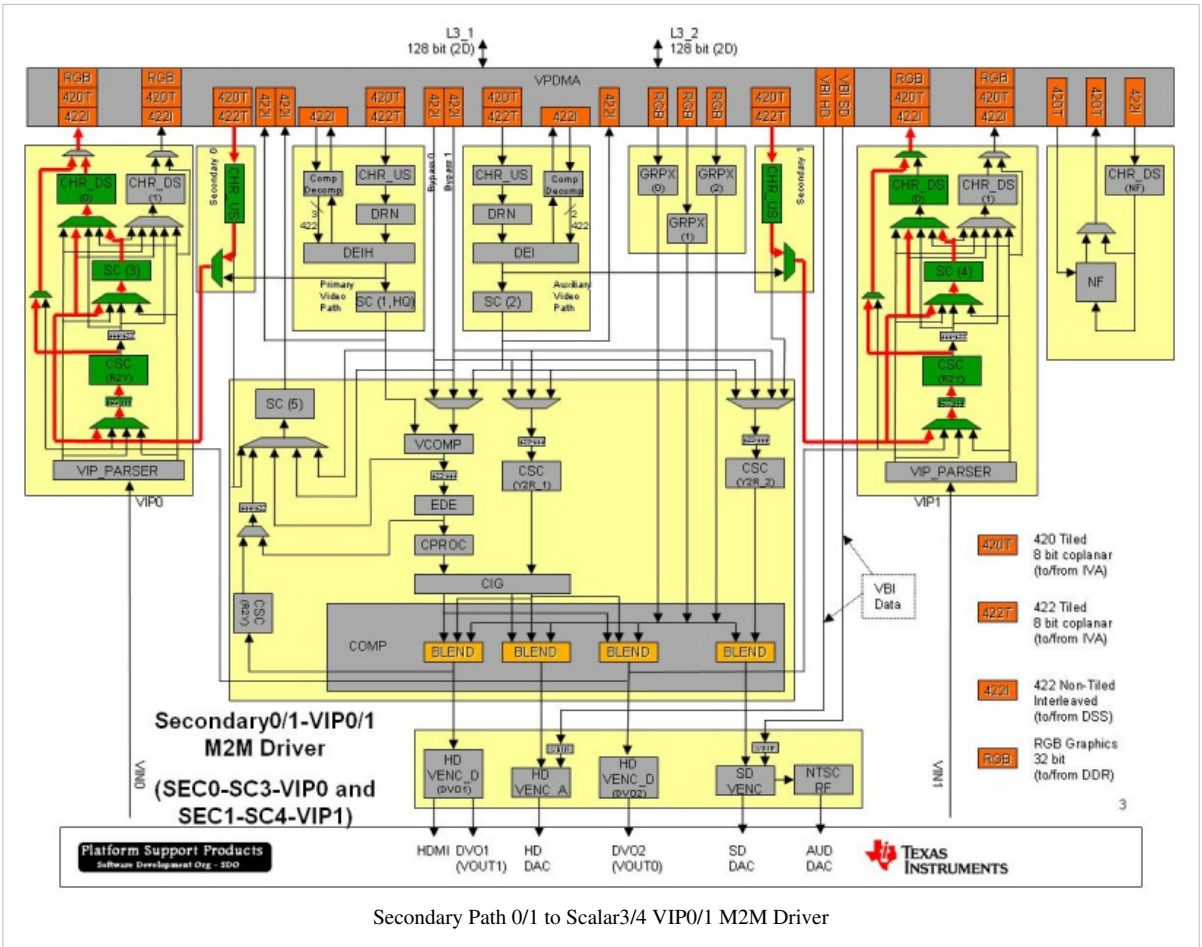
Below figure shows the different instances of the scalar driver. At a maximum 3 scalar drivers can be active at time.



Secondary Path 0 to Scalar 5 M2M Driver



Bypass Path 0/1 to Scalar M2M Driver



Secondary Path 0/1 to Scalar3/4 VIP0/1 M2M Driver

Following are the instances supported by the scalar driver.

VPS_M2M_INST_SEC0_SC5_WB2: This instance involves chroma upsampler in the secondary path and the scalar 5 hardware.

VPS_M2M_INST_BP0_SC5_WB2/VPS_M2M_INST_BP1_SC5_WB2: These two instances involves the bypass path and the scalar 5 hardware.

VPS_M2M_INST_SEC0_SC3_VIP0/VPS_M2M_INST_SEC1_SC4_VIP1: These two instances involves the chroma upsampler in the secondary path and the VIP hardware.

Chroma upsampler takes the YUV420 semi planar(NV12) or YUYV422 image format as the inputs and bypass path can take YUYV image format. Chroma upsampler input can be from the raster based buffer or the tiled buffer and bypass path can take input from the raster based buffer only. Medium quality scalar can scale the images from 1/8x to the line size that is 2048 pixels. It involves different types of scalars like poly phase scalars, running average scalars. It can also do the optional cropping of the image and then do the scaling primarily known as digital zoom feature. It also supports non linear scaling like conversion of the 4:3 aspect ratio to 16:9 and vice versa. Details about the scalar capabilities can be found in the HDVPSS specifications.

Instances supported

Instances	TI816x	TI814x/TI8107
VPS_M2M_INST_SEC0_SC5_WB2	Supported	Supported
VPS_M2M_INST_BP0_SC5_WB2/ VPS_M2M_INST_BP1_SC5_WB2	Supported	Supported

Features supported

Features Supported	VPS_M2M_INST_SEC0_SC5_WB2 Instance	VPS_M2M_INST_BP0_SC5_WB2/ VPS_M2M_INST_BP1_SC5_WB2 Instances	VPS_M2M_INST_SEC0_SC3_VIP0/ VPS_M2M_INST_SEC1_SC4_VIP1 Instances
Chroma up sampling from YUV420 semiplanar to YUYV422 interleaved format.	Supported	Not supported	Supported
Input from Tiled buffer	Supported	Not supported	Supported
Scaling from 1/8x to 2014 maximum pixels in horizontal direction. Vertical scaling upto 1080 lines without any ratio limitation.	Supported	Supported	Supported
Supports user programmable as well as standard set of coefficients for the scalar.	Supported	Supported	Supported
Supports horizontal and vertical cropping of the image before scaling.	Supported	Supported	Supported
Supports different types of scalar like poly phase and running average.	Supported	Supported	Supported
Output data formats supported	FVID2_DF_YUV422I_YUYV	FVID2_DF_YUV422I_YUYV	FVID2_DF_YUV422I_YUYV, FVID2_DF_YUV420SP_UV

Features Not Supported

- Interlaced image at input or output not supported.

Software Application Interfaces

The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: Here driver is used to submit the frames for processing and getting the processed frames from the driver.
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is de-initialized

The subsequent sections describe each phase in detail.

Note
Details of the structure, enumerations and #defines mentioned in the section can be found in HDVPSS API Guide

System Init Phase

The scalar driver initialization happens as part of overall HDVPSS system init. This API must be the the first API call before making any other FVID2 calls. Below section lists the APIs which are part of the System Init phase.

FVID2 Init

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

Create Phase

In this phase user application opens or creates a driver instance. Each instance of the driver supports VPS_M2M_SC_MAX_HANDLE (defined in vps_m2mSc.h) handles creation. Operation commands from the different handles of the same instance will be serialized by the driver and will be served by the singe instance of the hardware. Below sections lists the API interfaces to be used in the create phase. Create phase allows the application to do the configuration either through control commands exposed by driver or through the parameters passed with the driver create API.

FVID2_create

```
FVID2_Handle FVID2_create(UInt32 drvId,
                  UInt32 instanceId,
                  Ptr createArgs,
                  Ptr createStatusArgs,
                  const FVID2_CbParams *cbParams);
```

This API is used to open the driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

drvId - Driver Id for all the instances of the scalar driver is `FVID2_VPS_M2M_SC_DRV`

instanceId - Pass `VPS_M2M_INST_BP0_SC5_WB2/VPS_M2M_INST_BP1_SC5_WB2` macro to open the 422P bypass path1/bypass path2 instance of the driver. `VPS_M2M_INST_SEC0_SC5_WB2` opens secondary path scalar instance of driver and `VPS_M2M_INST_SEC0_SC3_VIP0/VPS_M2M_INST_SEC1_SC4_VIP1` opens the

secondary path scalar driver involving one of the VIP hardware.

createArgs - Pass a pointer to `Vps_M2mScCreateParams` structure containing the valid parameters. This parameter should not be null

createStatusArgs - Pass a pointer to `Vps_M2mScCreateStatus` structure.

cbParams - Pass the pointer to `FVID2_DrvCbParams`. These call back parameters are used to indicate the successful processing of the frame or the error frames.

FVID2 Control - Set Scalar Coefficient

This is used to issue a control command to the driver. `IOCTL_VPS_SET_COEFFS` ioctl is used to set the scalar coefficients. This is a blocking call.

Important
This API must not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SET_COEFFS` ioctl.

cmdArgs - Pointer to `Vps_ScCoeffParams` structure containing valid scaling coefficient. This parameter should not be NULL. Since this driver has a single scalar in all the paths, the `scalarId` can be set to 0. It is ignored.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

More information

This IOCTL is used to program the scalar coefficients. Scalar coefficients programmed by a handle will affect all the open handles and can be programmed only if none of the requests for that handle are pending to be processed in the driver. It supports standard set of coefficients for the different scaling ratios as well as the user coefficient

FVID2 Control - Enable/Disable Lazy Loading

This is used to issue a control command to the driver. `IOCTL_VPS_SC_SET_LAZY_LOADING` ioctl is used to enable/disable Scalar Lazy Loading. This is a blocking call.

Important
This API should not be called when there are any pending requests with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SC_SET_LAZY_LOADING` ioctl.

cmdArgs - Pointer to `Vps_ScLazyLoadingParams` structure. This parameter must not be NULL. Since this driver has a single scalar in all the paths, the `scalarId` can be set to 0. It is ignored.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

cmdStatusArgs - This is for future use. Application can safely pass NULL. Driver ignores this.

More information

This IOCTL enables or disables Lazy Loading for the scalar coefficients.

Without the Scalar Lazy Loading feature for memory to memory drivers, the user needs to set coefficients for all scalars in the driver instance as per the channel characteristics. When multiple drivers are simultaneously used, this makes programming more complex. By using Lazy Loading, the user does not need to be concerned with this, and can simply enable Lazy Loading, and the coefficient configuration happens automatically internally. Also, for multi-channel scaling, the scaler coefficients and filter type are fixed for the frames being scaled on all channels. Due to this, if frames of different size are provided on the channels, it results in lower quality scaling output. To ensure good quality scaling output, the user is hence required to issue frames one by one and make IOCTL_VPS_SET_COEFFS commands to change the scaling coefficients in between the issued frames. This causes programming complexity. When Lazy Loading is enabled, the driver internally configures the scalar coefficients if required, as per the frame characteristics, when the frames are issued, and the user does not need to configure the coefficients in between frame issues.

Features:

1. Scalar Lazy Loading is supported for memory-to-memory drivers only
    1. SC5: sec0 wb path
    2. SC1 and SC2: DEI path
    3. SC3 and SC4: VIP path
2. The driver internally selects the appropriate scaling coefficients for each frame (channel). If the scaling factor is different from the current scaling factor for that scalar, it internally sets the coefficients before processing the frame.
3. The driver internally selects the appropriate vertical scaling filter (polyphase or running average) for each frame depending on the scaling ratio for that frame. For higher than 1/4 scaling ratio for Vertical scaling, RAV filter is used.
    1. The polyphase filter is used for upscaling for horizontal as well as on vertical side.
    2. For horizontal scaling, decimation filters are internally configured when horizontal scaling factor is less than 1/2.
4. The decision to enable/disable lazy loading is configurable through the IOCTL: IOCTL_VPS_SC_SET_LAZY_LOADING
5. If a scalar is in bypass, then loading any coefficients does not happen for that scalar.
    1. If the src and dest are same, but scalar is not in bypass, the loading of coefficients still happens if there is a difference with previous coefficients.
6. If RT params are provided, the scaling factor configuration accordingly changes, and coefficient configuration may happen if necessary.
7. By default, Scalar Lazy Loading is disabled for all scalars.

Run Phase

M2m drivers are non-streaming drivers. This phase is used to submit the requests for processing and getting the processes request back.

Start and stop

NA

FVID2_processFrames

```
Int32 FVID2_processFrames(FVID2_Handle handle,
                  FVID2_ProcessList *processList);
```

This is used to submit the frames for processing. Processlist contains the frames from various channels to be processed. Application can open the driver for N channels and it can submit request for the M channels where M <= N. Processlist is returned to the driver and it can use the processlist for submitting the next request whereas all the elements inside the processlist is driver's ownership and can be reused by application only after de-queuing the request. This is a non blocking call and requests are queued inside the driver for processing. Driver calls the application call back once the hardware completes processing the request.

handle - Pass the handle returned to the application while opening of the driver.

processList - Pass the processlist containing the frames to be processed. User can also pass the run time parameters with the processlist to change the parameters run time. Run time parameters only needs to be passed once. Subsequent process_frames function will be operated with the last run time parameters passed. If user wants to updated that again new set of run time parameters needs to be passed. The driver supports run time parameters change in synchronous with the submitted frame. Run time parameter structure supported by scalar driver is `Vps_M2mScRtParams`.

Application needs to pass this structure either with the framelist or with the individual frames. If the driver is open with the same configuration for all the channels then user needs to pass the pointer to this structure with the input frame list or else if the driver is open with separate configuration for each channel user needs to pass the the pointer to this structure with each individual frame of the input frame list.

User need to populate the structure and pass the same pointer to inFramelist or the frames inside the frame list depending upon how the driver is opened as explained earlier.

Warning
Run time parameter interface may change in the future as this interface is currently experimental.

FVID2_getProcessedFrames

```
Int32 FVID2_getProcessedFrames(FVID2_Handle handle,
                       FVID2_ProcessList *processList,
                       UInt32 timeout);
```

This is used to de-queue the already processed request. This is again a non blocking call and if there are requests in the driver to be de-queued it will return the dequeued frames in the processlist else it will return with error.

handle - Pass the handle returned to the application while opening of the driver.

processList - Pass the pointer to the processlist. Driver will copy the pointer to the processed framelist to the processlist.

timeout - Currently unused arguments. Driver ignores this.

Delete Phase

In this phase FVID2 delete API is called to close the driver handle. Hardware resources are freed once all the handles of the particular instance are freed. Handle can be opened again once close with different configuration.

FVID2_delete

```
Int32 FVID2_delete(FVID2_Handle handle, Ptr deleteArgs);
```

This API is used to close the driver. This is again a blocking call. Call returns only when the handle is closed. This API will return error is the request is queued and application tries to close the driver. All queued requests needs to be de-queued before closing of the driver.

handle - Pass the handle returned to the application while opening of the driver.

deleteArgs - This is reserved for future use. Application must pass NULL.

System De-Init Phase

FVID2 de-Init

Drivers gets de-initializes as a part of HDVPSS sub-system de-Initialization. Here all resources acquired during system initialization are free'ed. Make sure all driver instances and handles are deleted before calling this API. Typically this is done during system shutdown.

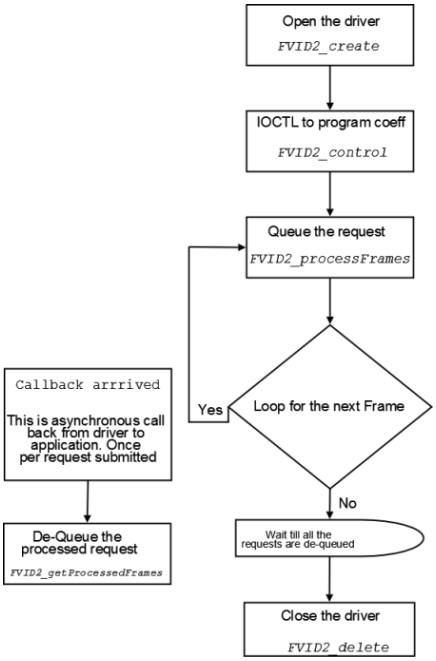
```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

Sample Application

Typical application flow

Following diagram shows the typical application flow for the driver:



Sample Application Flow - Secondary path to Scalar M2M Driver

Scalar driver application

Multichannel scalar applications exercise all the instances of the scalar driver. It presents the users with the different options to run scalar driver with different options and different configurations of the driver like multiple channels, different resolutions and different data formats. It also prints the frame per second achieved and the CPU load.

This example illustrates the five paths supported by SC5/SC3/SC4 scalar memory to memory driver for subframe based processing. Driver path can be selected using user input through console.

- Secondary 0 Path to SC5 Scale: VPS_M2M_INST_SEC0_SC5_WB2
- Bypass Path0 to SC5 Scale: VPS_M2M_INST_BP0_SC5_WB2
- Bypass Path1 to SC5 Scale: VPS_M2M_INST_BP1_SC5_WB2
- Secondary 0 Path to VIP SC3 Scale:VPS_M2M_INST_SEC0_SC3_VIP0
- Secondary 1 Path to VIP SC4 Scale:VPS_M2M_INST_SEC1_SC4_VIP1

Application takes the input buffers for processing, optionally processes the buffers based on the option selected and writes the buffer back to memory. Following are the steps to run the multichannel application.

- Please refer Common Steps for connecting CCS to TI816x, running gel file etc.
- Load *hdvpss_examples_m2mScMultiChan.xem3*' at *$(rel_folder)\build\example-name\bin\ti816x-evm\example-name-whole-program-debug.xem3*

to DSS M3 debug session

- Run the application.
- Different options are printed for the scalar driver on the console.
- Select the required options
- The application will halt for the user to load the input frames. Using `loadRaw` command load images on the memory location mentioned in the console print.

The command to be used for loading the image into the memory buffer shall be printed on the console. For example, the command for loading the images is similar to the below:

```
loadRaw(<addr>, 0,
"<filePath>\\<fileName>_nv12_prog_packed_720_480.tigf", 32,
 false);
```

- Enter an alphanumeric letter and press enter after loading
- Run the program till it shows the options again
- Save the processed images. Using `saveRaw` command, the image file can be saved. The command to be used for saving the image file from the memory buffer shall be printed on the console. For example, the command for saving the image file is similar to the below:

```
saveRaw(0, <addr>,
"<filePath>\\<fileName>_nv12_prog_packed_1920_1080.tigf",
777600, 32, true);
```

Warning

If the processor is not halted or waiting for console input when saving the images, the images will be seen green as the data dumped will be 0x00.

- View the saved images using any external YUV image viewer.
- The image can also be viewed with the Image Analyzer tool in CCSV4.
- Following image shows the properties of the image analyzer tool for viewing the image. Address of the each buffer needs to be changed and then right click on the image and say "Refresh".

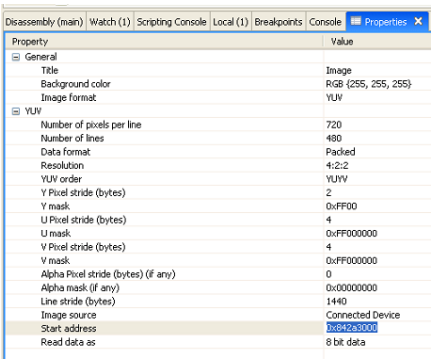


Image Analyzer properties for viewing 720X480 YUV Packed images

SubFrame Based Scaling

Introduction

Applications which requires low latency in video processing, like Video communications, divides video frame in to multiple parts, called subframes and do all processing at subframe level. This results in total end2end delay reduction and thus enhances user experience.

This chapter describes support of subframe based scaling in memory to memory drivers, application software interfaces, and sample application usage.

Software Overview

Application can enable subframe processing for each channel at Create time by setting enable flag and providing number of lines per subframe in Channel create parameters.

For Processing, Application should pass subframe number and number of lines available in the frame as input along with the frame start address, for each subframe. Driver will process this subframe and updates the number of lines available in output frame for further processing by other modules.

Chroma upsampler and scalar uses multi-tap filters to achieve their functionality. Because of this, subframe level processing requires few lines of video from previous and next subframes to match subframe level processing output with frame level processing. This memory is referred as Line memory.

Line memory required will change based on the type of vertical filter used in scalar, Polyphase/Running Average and input type, YUV420/YUYV422. This Line memory for each subframe is calculated internally by the driver and used for the subframe processing to adjust buffer offsets and other parameters. Also driver calculates scalar phase information for each subframe and program them in scalar for that subframe processing.

As each subframe contains integral number of lines, No special considerations are required for horizontal scalar.

Supported Drivers

SubFrame based processing is supported in following set of drivers:

- DEIH/DEI Memory to Memory Driver: MAIN-DEIH-SC1-WB0 and AUX-DEI-SC2-WB1 single scale paths.
- Scalar5 Memory to Memory Driver with Secondary 0 path OR Bypass path 0 OR Bypass path 1.

Features Supported

- SubFrame processing for different types of vertical scalar, like poly phase and running average.
- SubFrame processing for YUV420 semi planar(NV12) or YUYV422 progressive input.

Features Not Supported

- Interlaced image at input or output not supported.

Software Application Interfaces

This section describes SubFrame processing related structures and parameters setting in FVID2 level functions.

Application Interfaces

FVID2_create

```
FVID2_Handle FVID2_create(UInt32 drvId,
                  UInt32 instanceId,
                  Ptr createArgs,
                  Ptr createStatusArgs,
                  const FVID2_CbParams *cbParams);
```

This API is used to open the driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

To enable subframe processing, `Vps_SubFrameParams` structure elements inside `createArgs` structure of corresponding driver needs to be populated with below values:

- `Vps_SubFrameParams->subFrameModeEnable` should be set to TRUE
- `Vps_SubFrameParams->numLinesPerSubFrame` should be set to a value equal to framseSize/NumberOfSubFramesPerFrame.

FVID2_processFrames

```
Int32 FVID2_processFrames(FVID2_Handle handle,
                  FVID2_ProcessList *processList);
```

This is used to submit the frames for processing. Processlist contains the frames from various channels to be processed. Application can open the driver for N channels and it can submit request for the M channels where M <= **N. Processlist is returned to the driver and** it can use the processlist for submitting the next request whereas all the elements inside the processlist is driver's ownership and can be reused by application only after de-queuing the request. This is a non blocking call and requests are queued inside the driver for processing. Driver calls the application call back once the hardware completes processing the request.

handle - Pass the handle returned to the application while opening of the driver.

processList - Pass the processlist containing the frames to be processed.

For channels where subframe processing is enabled, `FVID2_SubFrameInfo` structure (inside `FVID2_Frame`) of the frames related to this channel needs to be populated with below values.

- `FVID2_SubFrameInfo->subFrameNum` should be set to '0' for first subframe and needs to be incremented by '1' for subsequent subframes.
- `FVID2_SubFrameInfo->numInLines` should be set to Number of lines avaialble in the frame at the end of current subframe.

Warning
SubFrame parameter interface may change in future as this interface is currently experimental.

FVID2_getProcessedFrames

```
Int32 FVID2_getProcessedFrames(FVID2_Handle handle,
                        FVID2_ProcessList *processList,
                        UInt32 timeout);
```

This is used to de-queue the already processed request. This is again a non blocking call and if there are requests in the driver to be de-queued it will return the dequeued frames in the processlist else it will return with error.

handle - Pass the handle returned to the application while opening of the driver.

processList - Pass the pointer to the processlist. Driver will copy the pointer to the processed framelist to the processlist.

For the channels where subframe processing is enabled, `FVID2_SubFrameInfo` structure of output frames will be updated by driver.

- `FVID2_SubFrameInfo->subFrameNum` contains latest subframe number processed by driver, 0 to N-1.
- `FVID2_SubFrameInfo->numOutLines` contains Number of lines avaialble in output frame after processing current subframe.

timeout - Currently unused arguments. Driver ignores this.

Sample Applications

SubFrame Processing in SC5 M2M driver

This example illustrates the three paths supported by SC5 scalar memory to memory driver for subframe based processing. Driver path can be selected using user input through console.

- Secondary 0 Path to SC5 Scale: VPS_M2M_INST_SEC0_SC5_WB2
- Bypass Path0 to SC5 Scale: VPS_M2M_INST_BP0_SC5_WB2
- Bypass Path1 to SC5 Scale: VPS_M2M_INST_BP1_SC5_WB2

Application opens the driver in a single channel configuration with one channel per handle.

Application takes the single YUV420 semi planar (NV12 format) image (SEC0 path) OR YUYV422 (for BP0/1 path) from the memory of size SD (720X480) as 4 subframes and outputs 1920x1080 image of YUYV422 format. For SEC0 pathm Chroma upsampler in the path converts the YUV420 to YUYV22 while the scalar in the path scales the image from SD to full HD. Following are the steps to run the sample application.

- Please refer Common Steps for connecting CCS to TI816x, running gel file etc.
- Load *hdvpss_examples_m2mScMultiChan.xem3*' at *$(rel_folder)\build\bin\$platform\m3vpss\whole_program_debug*

to DSS M3 debug session

- Run the application.
- The application will halt for the user to select driver instance to run and to load the input frames.

- Please select the any one of the options for subframe based scaling examples:
- 1 CH SUB-FRAME PROCESSING YUV420 SD ---> YUV422 HD, Driver: VPS_M2M_INST_SEC0_SC5_WB2
- 1 CH SUB-FRAME PROCESSING YUV422 SD ---> YUV422 HD, Driver: VPS_M2M_INST_BP0_SC5_WB2
- 1 CH SUB-FRAME PROCESSING YUV422 SD ---> YUV422 HD, Driver: VPS_M2M_INST_BP1_SC5_WB2
- Using loadraw command load the YUV420 Semiplanar/YUYV422 image of size 720X480 on the memory location mentioned in the console print. Following is the command for loading the image.

```
loadRaw(<addr>,0,"d:\\NV_12\\000_nv12_prog_packed_720_480.tigf",32,false);
```

- Enter an alphanumeric letter and press enter after loading
- Run the program till it outputs "Test Successful!!" on console
- Halt the processor and save the image. Following is the command for saving the image. Program prints the output buffer address. Please verify that output buffer address are same from where the image is stored. If its not same save the images from correct address after modifying the below addresses.

```
saveRaw(0,<addr>,"d:\\results\\ch.yuv",1036800,32,true);
```

Warning
If the proccessor is not halted before saving of the images. Images will be seen green as the data dumped will be 0x00.

- View the image with the Image Analyzer tool in CCSV4. Image format is YUYV422 interleaved 1920X1080
- Saved Images can also be viewed using any external YUV image viewer.

SubFrame processing in DEI M2M Single Scale

This example illustrates two paths supported by DEI memory to memory driver for subframe processing. Driver path can be selected using user input through console.

- DEIH Single Scale writeback: MAIN-DEIH-SC1-WB0
- DEI Single Scale writeback: AUX-DEI-SC2-WB1

DEIH/DEI Single Scale SubFrame processing example features: 720x480 progressive YUV420 data is fed into the DEI path. DEI, DRN are configured in bypass mode. This application will feed the frame as 4 subframes to driver. Input is scaled to 360x240 (YUV422) in 4 parts and written to memory via the WB0/WB1 path.

- Please refer Common Steps for connecting CCS to TI816x, running gel file etc.

Warning
Please recompile hdvpss_example_m2mDeiScale.xem3 after editing packages\ti\psp\examples\ti816x\vps\m2m\m2mDeiScale\src\M2mDeiScale_test.c to define SC_APP_TEST_SUBFRAME and also reduce DEI_TOTAL_LOOP_COUNT macro to 1. This will be fixed in next release to provide pre-built executable

- Load *hdvpss_example_m2mDeiScale.xem3* executable file found at *$(rel_folder)\build\bin\ti816x-evm\m3vpss\whole_program_debug* to DSS M3 debug session
- Run the application
- The application will halt for the user to load the input frames and to select driver path. Using loadRaw command in script console of CCS, load one 720 x 480 YUV420 semi-planar video to location mentioned in the console print. (Ignore "syntax error" if it appears during loading)

```
loadRaw(< Location >, 0, " < File Path > ", 32, false);
```

- Choose the required mode of driver from displayed options and enter in 'CCS console' window

```
Ex: For DEI WB1 single output driver, type '1' followed by 'enter' in console
window.
```

- The application will print status on console for each subframe processed. After processing is complete, application will wait for the user to save scaled output to file. Use the output buffer address printed in console for 'Location' in below command.

```
saveRaw(0, < Location >, " < File Path > ", 436800, 32, true);
```

- Press any numeric key to exit application after saving output image.

Deinterlacer - Memory to Memory Driver

TI814X/TI8107 Deinterlacer Memory to Memory Driver

→ TI814X/TI8107 DEI M2M Driver

TI816X Deinterlacer Memory to Memory Driver

→ TI816X DEI M2M Driver

UserGuideHdvpssTi816xDeiM2mDriver

TI816X Deinterlacer (DEIH/DEI) Memory to Memory Driver

Introduction

This chapter describes the hardware overview, application software interfaces, typical application flow and sample application usage for DEIH/DEI memory to memory driver.

The features and limitations of current driver implementation are listed in subsequent sections.

Important

The features supported or NOT supported in any release of the driver may vary from one HDVPSS driver release to another. See respective release notes for exact release specific details.

Features Supported

Features	Supported in TI816x
Instances	
MAIN-DEIH-SC1-WB0 single scale paths	YES
AUX-DEI-SC2-WB1 single scale paths	YES
MAIN-DEIH-SC3-VIP0 dual scale paths	YES
AUX-DEI-SC4-VIP1 dual scale paths	YES
MAIN-DEIH-SC1-SC3-WB0-VIP0 dual scale paths	YES
AUX-DEI-SC2-SC4-WB1-VIP1 dual scale paths	YES
Input Formats	
YUV422 Interleaved	YES

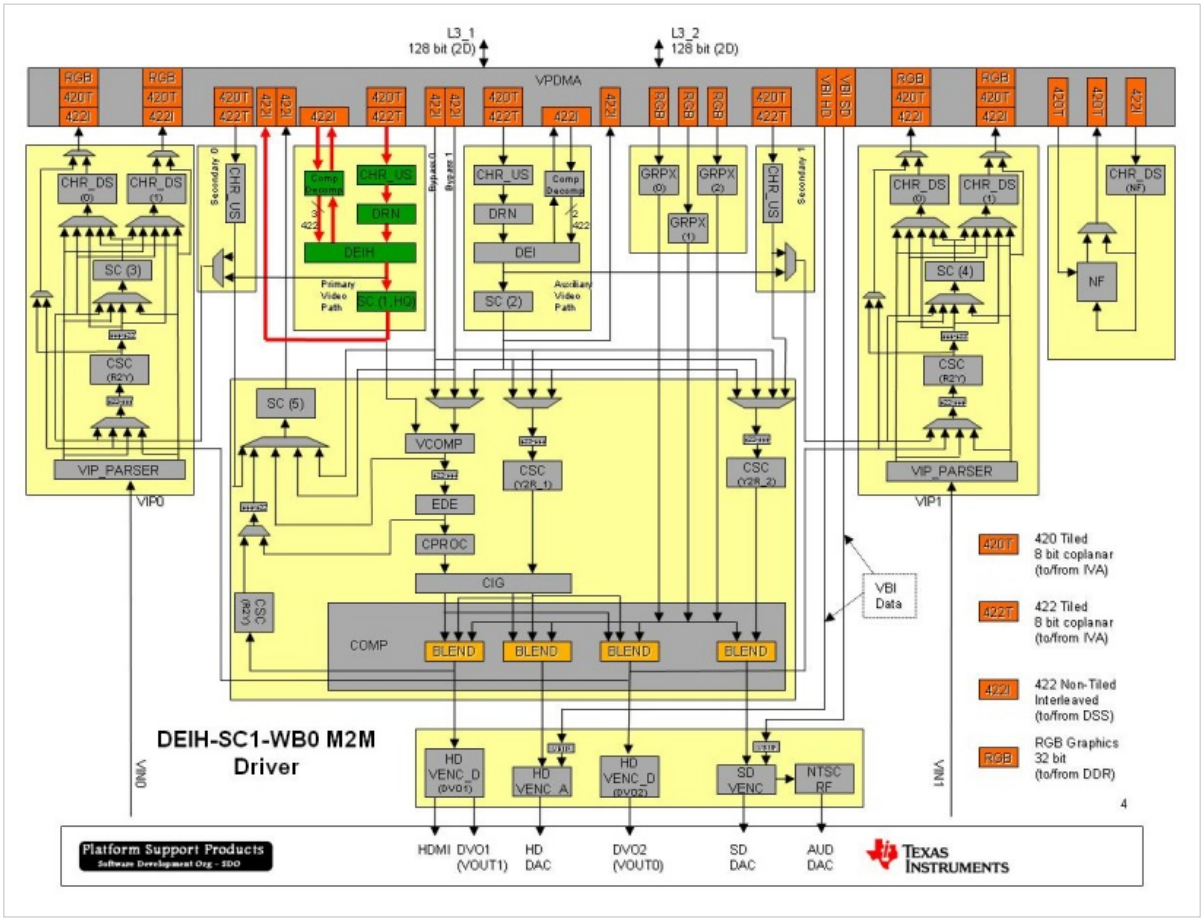
YUV420 Semi-Planar	YES
YUV422 Semi-Planar	YES
YUV420 Semi-Planar Tiled	YES
YUV422 Semi-Planar Tiled	YES
Output Formats	
YUV422 Interleaved on WB0/1	YES
YUV422 Interleaved on VIP0/1	YES
YUV420 Semi-Planar on VIP0/1	YES
YUV422 Semi-Planar on VIP0/1	YES
YUV420 Semi-Planar Tiled on VIP0/1	YES
YUV422 Semi-Planar Tiled on VIP0/1	YES
RGB on VIP0/1	NO
DEI Features	
DEI in deinterlacing mode	YES
DEI in progressive bypass mode	YES
Compression enable/disable for previous field inputs	YES
DEIH/DEI Mode1 in which both DEIH and DEI are operating to provide 30 fps output from DEIH from 60 input fields per second	YES
Line averaging and field averaging mode of DEI operation	YES
Progressive TNR operation in DEIH	YES
SC Features	
Optional scaling using SC1, SC2, SC3 and SC4	YES
Scaling from 1/8x to 2048 maximum pixels in horizontal direction	YES
Different types of scalar like poly phase and running average	YES
Horizontal and vertical cropping of the image before scaling	YES
User programmable scalar coefficients	YES
Other Features	
Enable/disable of DRN	YES
Frame drop feature on WB0/1 and VIP0/1 outputs to enable load balancing	YES
Multi-channel (up to `VPS_M2M_DEI_MAX_HANDLE_PER_INST` channels per instance)	YES
Multi-handle (up to `VPS_M2M_DEI_MAX_HANDLE_PER_INST` channels per instance)	YES
Error callbacks	YES
Slice based scaling when DEI is in progressive bypass mode	YES
Slice based scaling when DEI is in deinterlacing mode	NO
Interlaced bypass mode	NO
Runtime Configurations	
Input resolution change when DEI is in progressive bypass mode	YES
Input resolution change when DEI is in deinterlacing mode	YES
Output resolution change on WB0/1	YES
Output resolution change on VIP0/1	YES

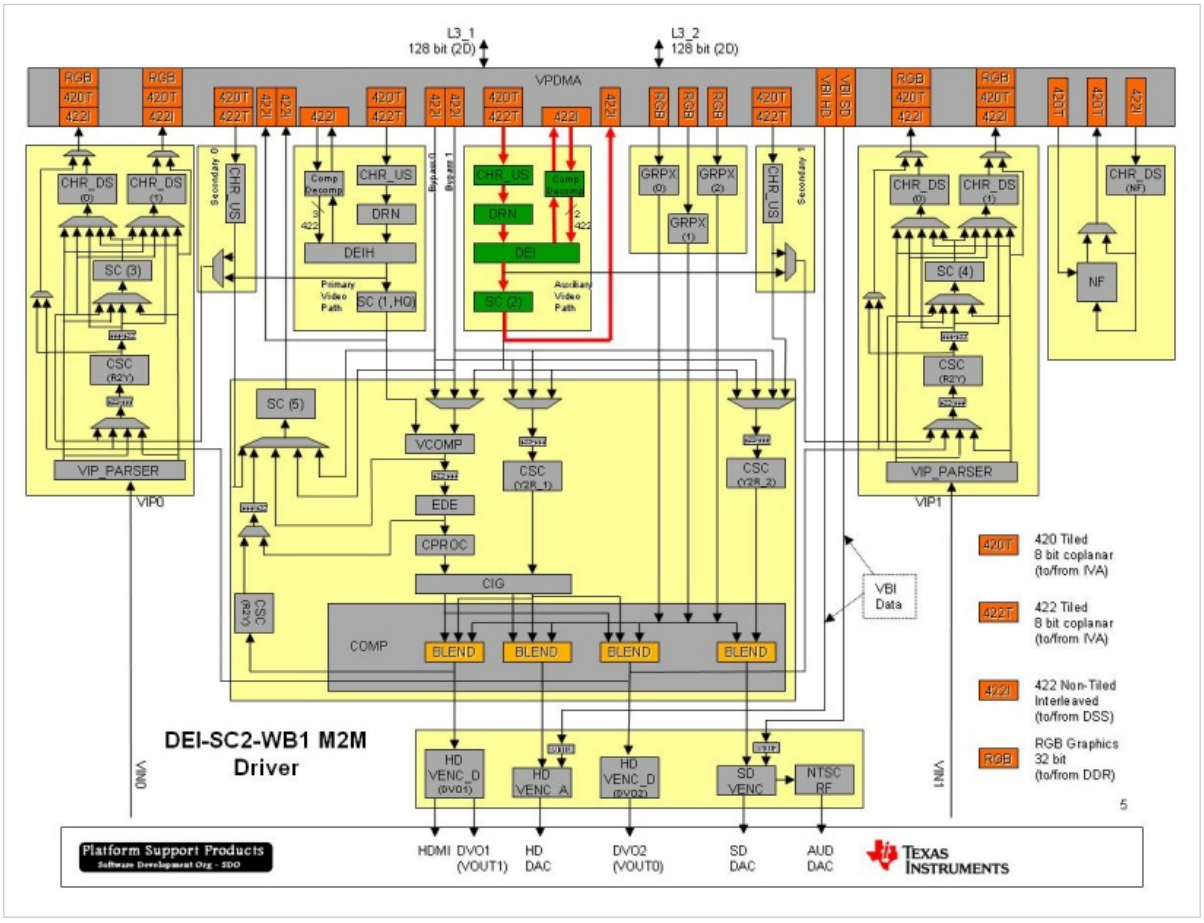
SC crop and config change on SC1/SC2	YES
SC crop and config change on SC3/SC4	YES
DEI reset	YES

Hardware Overview

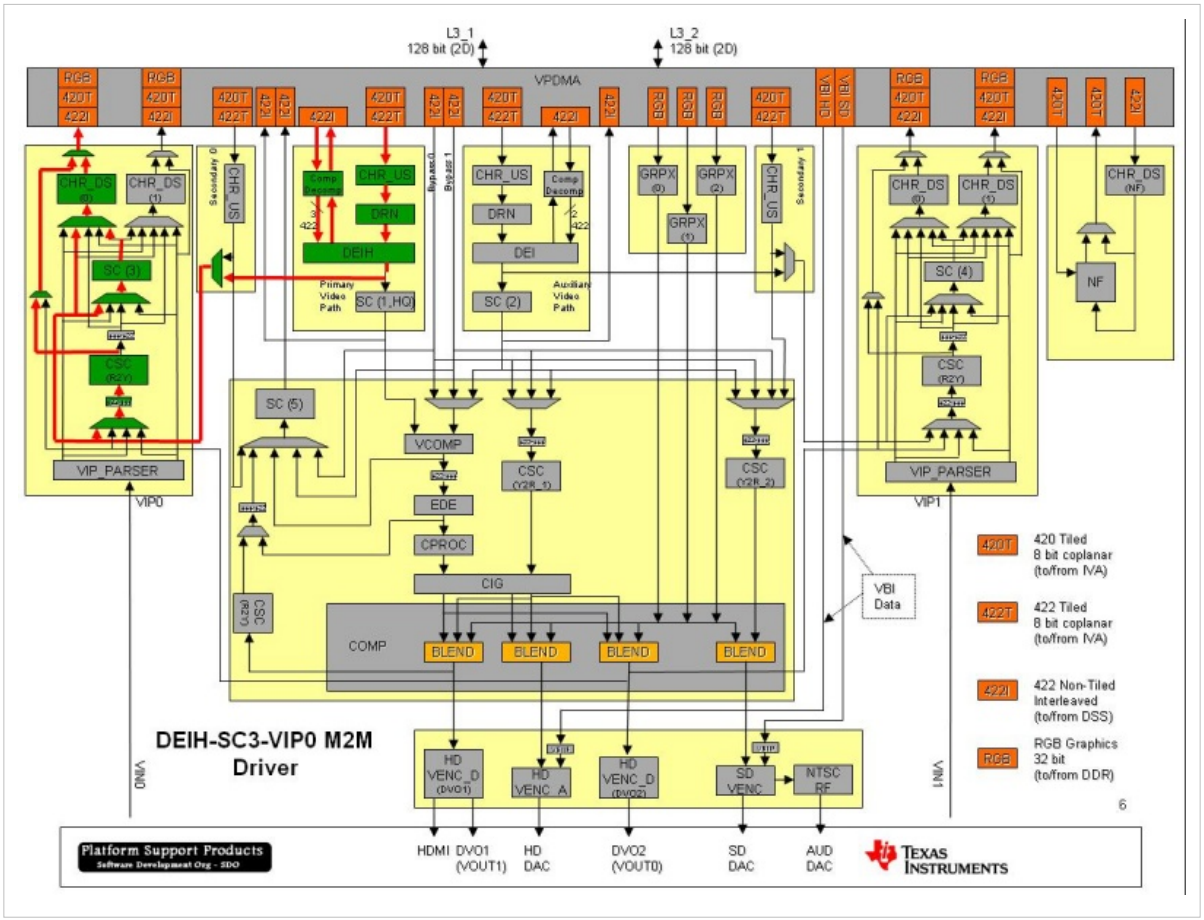
Below figures show the complete HDVPSS Hardware. The red bold lines in the figure shows the path on which the DEI memory to memory driver operates.

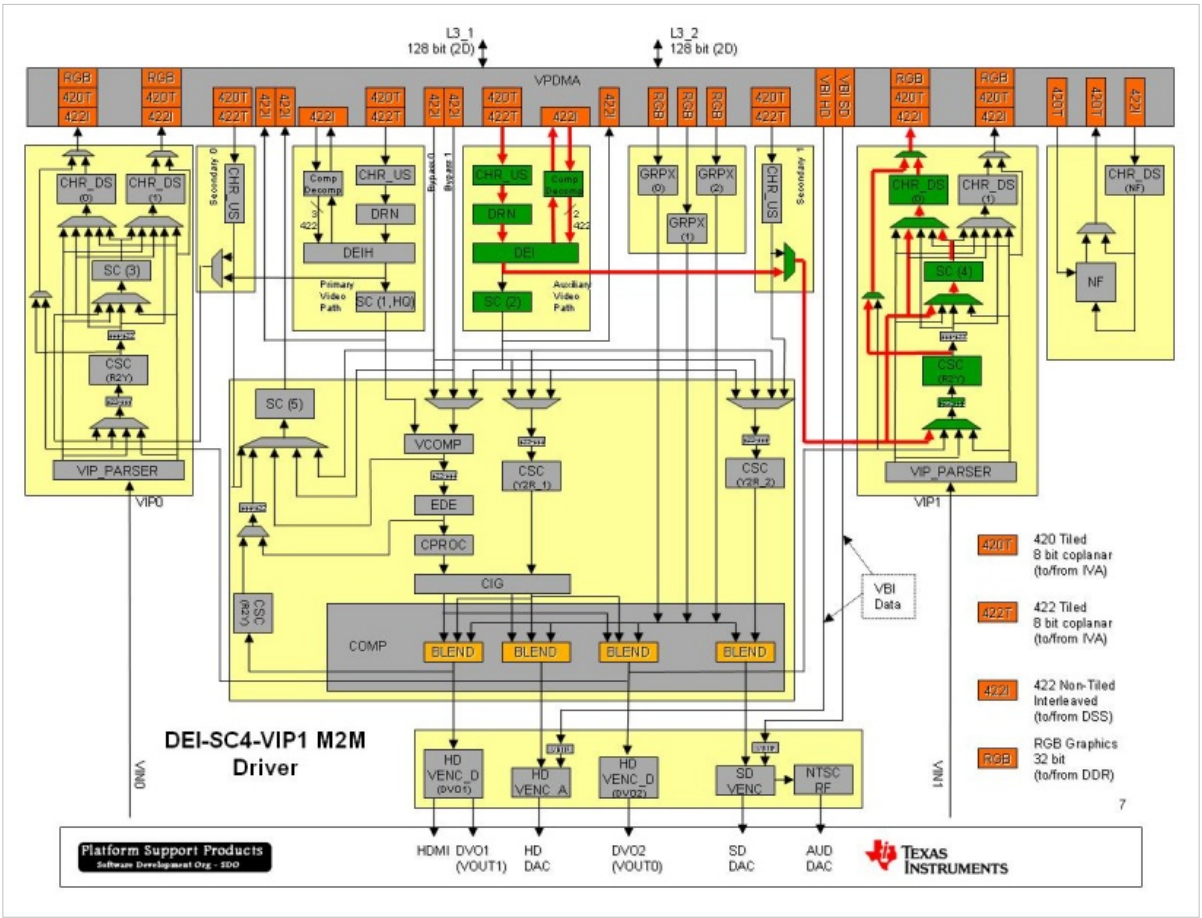
DEI-WB0/1 Single Output Paths: As shown in below figures, the DEI memory to memory driver takes in YUYV422/YUV420 interlaced/progressive input via the DEI path and provide single scaled output of the deinterlaced/bypassed output via writeback path 0/1.



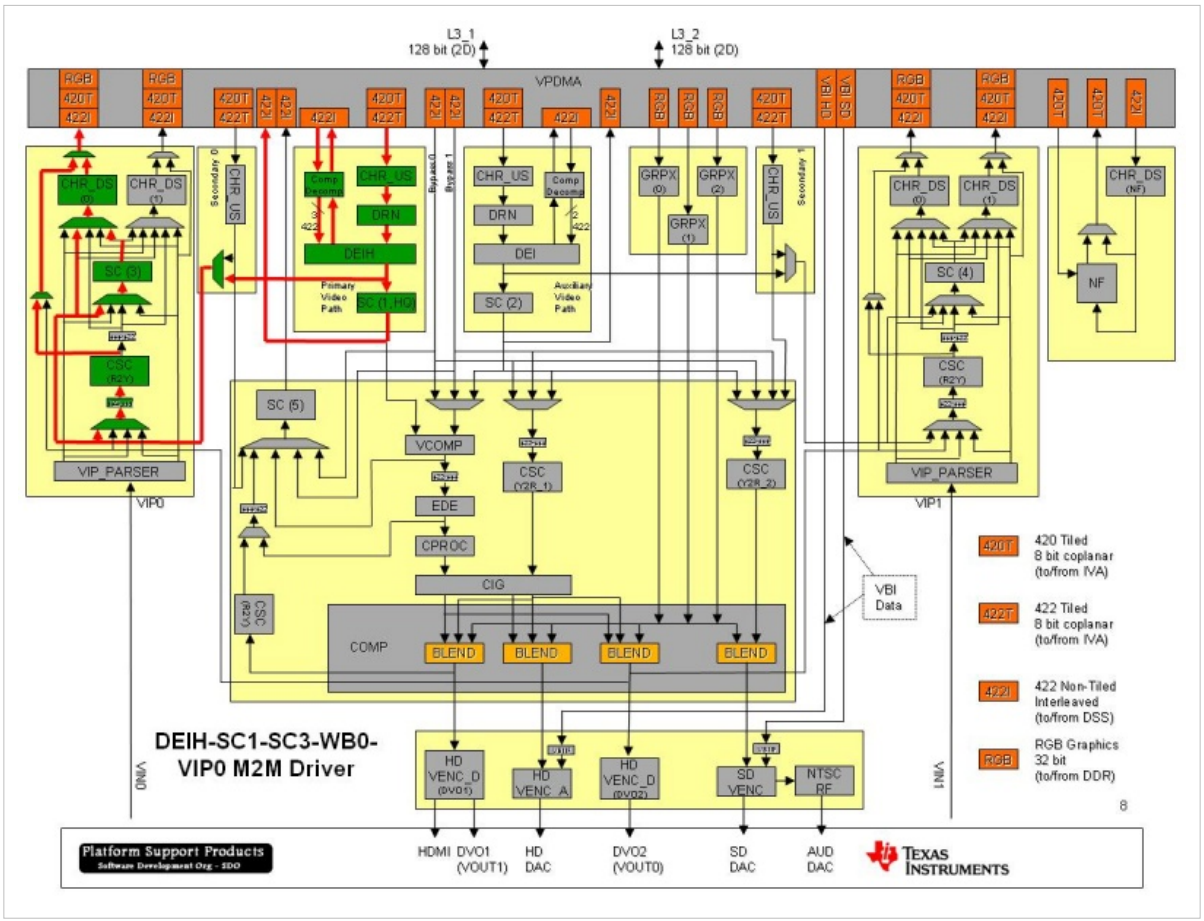


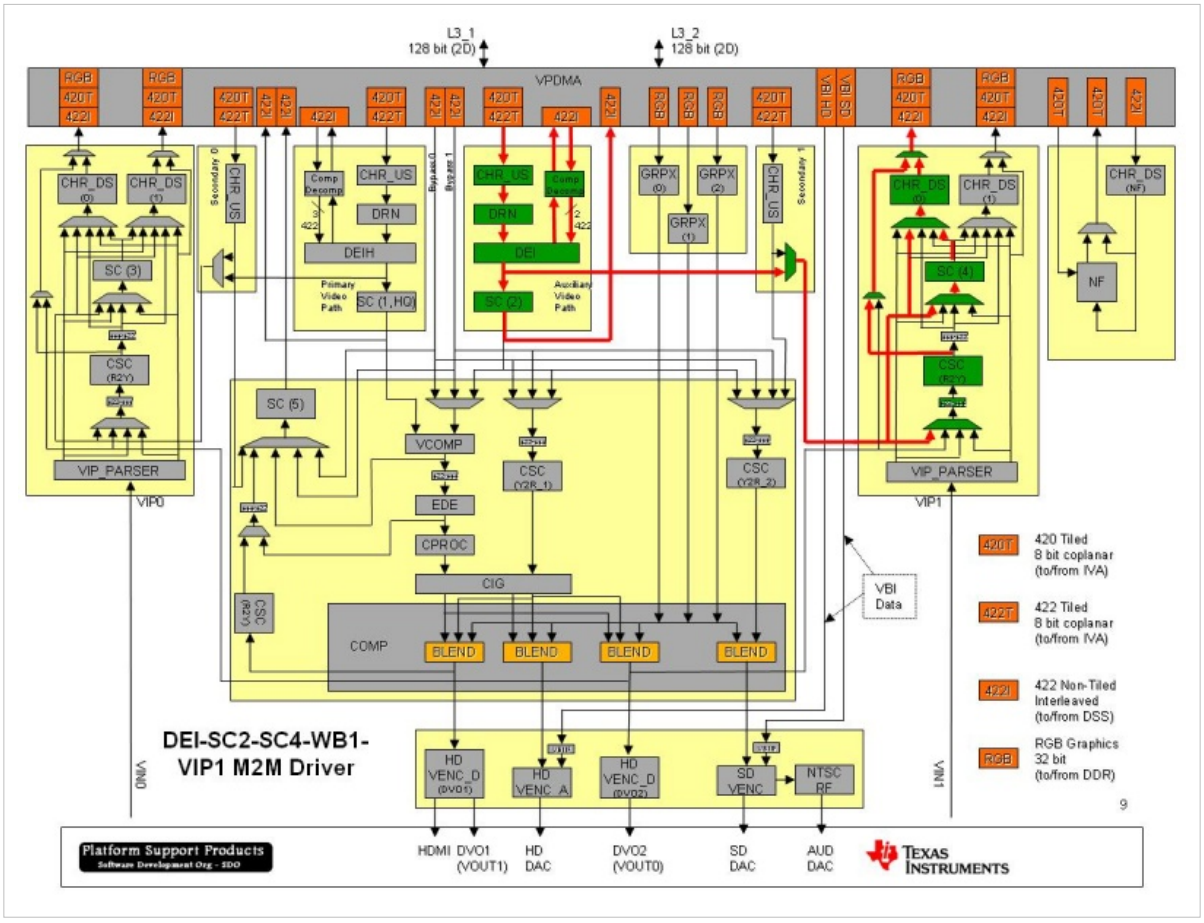
DEI-VIP0/1 Single Output Paths: As shown in below figures, the DEI memory to memory driver takes in YUYV422/YUV420 interlaced/progressive input via the DEI path and provide single scaled output of the deinterlaced/bypassed output via VIP path 0/1.


L3_1
128 bit (2D)
L3_2
128 bit (2D)
VPDMA
CHR_DS
SC (3)
CSC
VIP_PARSER
VIP0
CHR_US
DEIH
SC (1,HQ)
DRN
DEI
SC (2)
GRPX
SC (5)
VCOMP
CSC
EDE
CPROC
CIG
COMP
BLEND
SC (4)
NF
VIP1
VBI Data
DEIH-SC3-VIP0 M2M Driver
HD VENC_D (DVO1)
HD VENC_A
HD VENC_D (DVO2)
SD VENC
NTSC RF
HDMI
DVO1 (VOUT1)
HD DAC
DVO2 (VOUT0)
SD DAC
AUD DAC
420T 420 Tiled 8 bit coplanar (to/from IVA)
422T 422 Tiled 8 bit coplanar (to/from IVA)
422I 422 Non-Tiled Interleaved (to/from DSS)
RGB RGB Graphics 32 bit (to/from DDR)
Platform Support Products
Texas Instruments

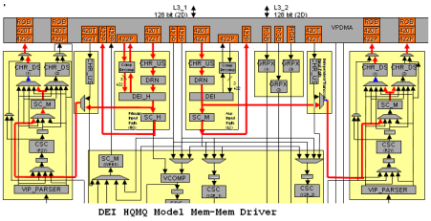


MAIN-DEIH-SC1-SC3-WB0-VIP0 and AUX-DEI-SC2-SC4-WB1-VIP1 Dual Output Paths: As shown in below figures, the DEI memory to memory driver takes in YUYV422/YUV420 interlaced/progressive input via the DEI path and provide two scaled version of the deinterlaced/bypassed outputs - one via writeback path 0/1 and another via VIP 0/1.


L3_1
128 bit (2D)
L3_2
128 bit (2D)
VPDMA
CHR_DS
SC (3)
CSC
VIP_PARSER
VIP0
SC (5)
VCOMP
EDE
CPROC
CIG
COMP
BLEND
CSC
DEIH-SC1-SC3-WB0-
VIP0 M2M Driver
DRN
DEI
SC (2)
SC (1_HQ)
GRPX
CHR_US
SC (4)
NF
VIP1
VBI
Data
HD
VENC_D
HD
VENC_A
SD
VENC
NTSC
RF
VIP0
VIP1
420T
420 Tiled
8 bit coplanar
(to/from IVA)
422T
422 Tiled
8 bit coplanar
(to/from IVA)
422I
422 Non-Tiled
Interleaved
(to/from DSS)
RGB
RGB Graphics
32 bit
(to/from DDR)
Platform Support Products
HDMI
DVO1
(VOUT1)
HD
DAC
DVO2
(VOUT0)
SD
DAC
AUD
DAC
Texas
Instruments



Below figure shows the DEIH-DEI Mode1 driver. The red bold lines in the figure shows the path on which the DEIH-DEI Mode1 memory to memory driver operates.



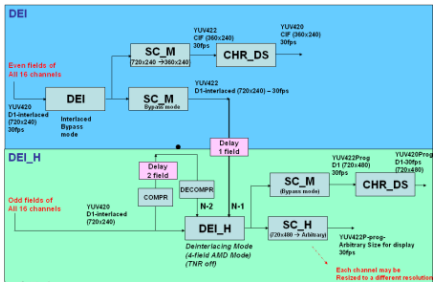
HDVPSS Block Diagram - DEIH-DEI Mode1 Memory Path

In this mode of DEI driver, both the DEIH and DEI are used to deinterlace input fields. DEI is used to get YUYV422 interleaved data from YUV420 semi-planar input data. This YUYV422 interleaved output will be fed to the DEIH as one field delayed data. DEIH is used in 4 field mode to deinterlace input fields. It takes two input fields i.e. current field and previous field from DEI and generates deinterlaced frame.

Even field of all the input channels are fed to the DEI. DEI will provide YUYV422 interleaved data from WB1 and it will provide scaled YUV420 frames from VIP1. The odd field of the all the input channels are fed to the DEIH along with the 422 interleaved fields from DEI to deinterlace these fields.

Below diagram shows an example of DEIH-DEI Mode1 driver where even fields of all the input channels are fed to DEI to get the YUYV422 interleaved data and these YUYV422 interleaved data is fed to the DEIH as one field data along with the odd fields of all the input channels. DEIH deinterlaces odd fields of all the channels and generates frame. This generated frame can again be scaled in VIP0 to get the YUYV422 interleaved or YUV420 semi-planar data. It can also be scaled in SC1 to get the YUYV422 interleaved data.

Since only one field of the input channel is getting deinterlaced, output frame rate will be 30 frames per second from 60 fields per second for all the input channels in this mode of the driver.



HDVPSS Block Diagram - DEIH-DEI Mode1 Path

Software Application Interfaces

The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: Here driver is used to submit the frames for processing and getting the processed frames from the driver.
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is de-initialized

The subsequent sections describe each phase in detail.

Note
Details of the structure, enumerations and #defines mentioned in the section can be found in HDVPSS API Guide

System Init Phase

DEI M2m driver initialization happens as part of overall HDVPSS system init. This API must be the the first API call before making any other FVID2 calls. Below section lists the APIs which are part of the System Init phase.

FVID2 Init

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

```
    /* Init FVID2 and VPS */
    retVal = FVID2_init(NULL);
    if (FVID2_SOK != retVal)
    {
        System_printf("FVID2 Init failed\n");
    }
```

Create Phase

In this phase user application opens or creates a driver instance. Each instance of the driver supports VPS_M2M_DEI_MAX_HANDLE_PER_INST (defined in vps_m2mDei.h) handles creation. Operation commands from the different handles of the same instance will be serialized by the driver and will be served by the singe instance of the hardware. Below sections lists the API interfaces to be used in the create phase. Create phase allows the application to do the configuration either through control commands exposed by driver or through the parameters passed with the driver create API.

FVID2 Create

This API is used to open the driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

```
FVID2_Handle FVID2_create(UInt32 drvId,
                          UInt32 instanceId,
                          Ptr createArgs,
                          Ptr createStatusArgs,
                          const FVID2_CbParams *cbParams);
```

drvId - `FVID2_VPS_M2M_DEI_DRV` to open the driver.

instanceId - `VPS_M2M_INST_MAIN_DEIH_SC1_SC3_WB0_VIP0` macro to open MAIN-DEIH-SC1-SC3-WB0-VIP0 dual scale memory driver. `VPS_M2M_INST_AUX_DEI_SC2_SC4_WB1_VIP1` macro to open AUX-DEI-SC2-SC4-WB1-VIP1 dual scale memory driver. `VPS_M2M_INST_MAIN_DEIH_SC1_WB0` macro to open MAIN-DEIH-SC1-WB0 single scale memory driver. `VPS_M2M_INST_AUX_DEI_SC2_WB1` macro to open AUX-DEI-SC2-WB1 single scale memory driver. `VPS_M2M_INST_MAIN_DEIH_SC3_VIP0` macro to open MAIN-DEIH-SC3-VIP0 single scale memory driver. `VPS_M2M_INST_AUX_DEI_SC4_VIP1` macro to open AUX-DEI-SC4-VIP1 single scale memory driver.

createArgs - Pointer to `Vps_M2mDeiCreateParams` structure containing valid create params. This parameter should not be NULL.

createStatusArgs - Pointer to `Vps_M2mDeiCreateStatus` structure containing the return value of create function and other driver information. This parameter should not be NULL.

cbParams - Pointer to `FVID2_CbParams` structure containing FVID2 callback parameters. This parameter should not be NULL.

```
    FVID2_Handle            fvidHandle;
    FVID2_CbParams          cbParams;
    Vps_M2mDeiChParams      chPrms;
    Vps_M2mDeiCreateParams  createParams;
    Vps_M2mDeiCreateStatus  createStatus;

    /* Init create params */
    createParams.mode = VPS_M2M_CONFIG_PER_CHANNEL;
    createParams.numCh = 1u;
    createParams.deiHqCtxMode = VPS_DEIHQ_CTXMODE_DRIVER_ALL;
    createParams.chParams = &chPrms;
    createParams.isVipScReq = TRUE;

    /* Init callback parameters */
    cbParams.cbFxn = App_m2mDeiAppCbFxn;
```

```
    cbParams.errCbFxn = App_m2mDeiAppErrCbFxn;
    cbParams.errList = &errProcessList;
    cbParams.appData = NULL;
    cbParams.reserved = NULL;

    /* Open the driver */
    fvidHandle = FVID2_create(
                    FVID2_VPS_M2M_DEI_DRV,
                    VPS_M2M_INST_MAIN_DEIH_SC1_SC3_WB0_VIP0,
                    &createParams,
                    &createStatus,
                    &cbParams);
    if (NULL == fvidHandle)
    {
        System_printf("Create failed!!\n");
    }
```

FVID2 Control - Set Scalar Coefficient

This is used to issue a control command to the driver. `IOCTL_VPS_SET_COEFFS` ioctl is used to set the scalar coefficients. This is a blocking call.

Important
This API should not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SET_COEFFS` ioctl.

cmdArgs - Pointer to `Vps_ScCoeffParams` structure containing valid scaling coefficient. This parameter should not be NULL. To set the scalar coefficient for DEI scalar, `scalarId` should be set to `VPS_M2M_DEI_SCALAR_ID_DEI_SC` and for VIP scalar `scalarId` should be set to `VPS_M2M_DEI_SCALAR_ID_VIP_SC`.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Enable/Disable Lazy Loading

This is used to issue a control command to the driver. `IOCTL_VPS_SC_SET_LAZY_LOADING` ioctl is used to enable/disable Scalar Lazy Loading. This is a blocking call.

Important
This API should not be called when there are any pending requests with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SC_SET_LAZY_LOADING` ioctl.

cmdArgs - Pointer to `Vps_ScLazyLoadingParams` structure. This parameter must not be NULL. To enable/disable Lazy Loading for DEI scalar, `scalarId` should be set to `VPS_M2M_DEI_SCALAR_ID_DEI_SC` and for VIP scalar `scalarId` should be set to `VPS_M2M_DEI_SCALAR_ID_VIP_SC`.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Get DEI Context Information

When DEI is in de-interlacing mode, DEI requires previous fields and motion vectors. Buffers for storing these context information must be allocated by the application and provided to the driver before starting M2M operation. `IOCTL_VPS_GET_DEI_CTX_INFO` ioctl is used to get the number of internal buffers to be allocated and their sizes. Application should get this information from the driver and allocate these buffers and provide the buffers to the driver before issuing any request. Once these buffers are given to the driver, application should not modify these buffers. This should be done for each and every channel. This is a blocking call.

Important
This API should not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_GET_DEI_CTX_INFO` ioctl.

cmdArgs - Pointer to `Vps_DeiCtxInfo` structure where the DEI context information will be filled by driver. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Set DEI Context Buffers

`IOCTL_VPS_SET_DEI_CTX_BUF` ioctl is used to set the DEI context buffers for a channel before providing any request to the driver. This is a blocking call.

Important
This API should not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SET_DEI_CTX_BUF` ioctl.

cmdArgs - Pointer to `Vps_DeiCtxBuf` structure valid buffer pointers as requested by the driver for a particular DEI mode of operation. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Get DEI Context Buffers

`IOCTL_VPS_GET_DEI_CTX_BUF` ioctl is used to get the DEI context buffers for a channel from the driver. Once the DEI context buffer is returned to the application, no more request should be provided to the driver. This is a blocking call.

Important
This API should not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_GET_DEI_CTX_BUF` ioctl.

cmdArgs - Pointer to `Vps_DeiCtxBuf` structure where the driver returns back the DEI context buffer to the application. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

Run Phase

M2m drivers are non-streaming drivers. This phase is used to submit the requests for processing and getting the processes request back.

Start and stop

NA

FVID2 Process Frames

This API is used to submit video buffers to the driver for processing operation. This is a non-blocking call and should be called from task context. Once the buffer is queued the application loses ownership of the buffer and is not suppose to modify or use the buffer.

```
Int32 FVID2_processFrames(FVID2_Handle handle,
                          FVID2_ProcessList *processList);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

processList - Pointer to `FVID2_ProcessList` structure containing the pointer to the FVID2 frames/framelist. This parameter should not be NULL.

FVID2 Get Processed Frames

This API is used by the application to get ownership of the processed video buffer from the memory driver. This is a non-blocking call and could be called from task or ISR context.

```
Int32 FVID2_getProcessedFrames(FVID2_Handle handle,
                               FVID2_ProcessList *processList,
                               UInt32 timeout);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

processList - Pointer to `FVID2_ProcessList` structure where the driver will copy the processed FVID2 frames/framelist. This parameter should not be NULL.

timeout - Not used currently as only non-blocking queue/dequeue operation is supported. This parameter should be set to FVID2_TIMEOUT_NONE.

Delete Phase

In this phase FVID2 delete API is called to close the driver handle. Hardware resources are freed once all the handles of the particular instance are freed. Handle can be opened again, once close, with different configuration.

FVID2 Delete

This API is used to close the memory driver. This is a blocking call and returns after closing the handle.

```
Int32 FVID2_delete(FVID2_Handle handle, Ptr deleteArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

deleteArgs - Not used currently. This parameter should be set to NULL.

System De-Init Phase

FVID2 de-Init

Drivers gets de-initializes as a part of HDVPSS sub-system de-Initialization. Here all resources acquired during system initialization are free'ed. Make sure all driver instances and handles are deleted before calling this API. Typically this is done during system shutdown.

```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

Sample Application

Sample Application Flow

Following diagrams show the typical application flows for the DEI memory driver - TODO

DEI Single and Dual Scale

This example illustrates the six paths supported by DEI memory to memory driver involving single and dual scaling outputs as listed below. Driver path can be selected using user input through console.

- DEIH Single Scale: MAIN-DEIH-SC1-WB0
- DEI Single Scale: AUX-DEI-SC2-WB1
- DEIH-VIP0 Single Scale: MAIN-DEIH-SC3-VIP0
- DEI-VIP1 Single Scale: AUX-DEI-SC4-VIP1
- DEIH-VIP0 Dual Scale: MAIN-DEIH-SC1-SC3-WB0-VIP0
- DEI-VIP1 Dual Scale: AUX-DEI-SC2-SC4-WB1-VIP1

DEIH/DEI Single Scale example features: 720x240 interlaced YUV420 data is fed into the DEI path. DEI is configured in deinterlacing mode. The deinterlaced input is scaled to 360x240 (YUV422) and output via the WB0/WB1 path.

DEIH/DEI-VIP0/1 Single Scale example features: 720x240 interlaced YUV420 data is fed into the DEI path. DEI is configured in deinterlacing mode. The deinterlaced input is is fed to the VIP0/1 through the transcode path and converted to 720x480 YUV420 progressive output.

DEIH-VIP0/DEI-VIP1 Dual Scale example features: 720x240 interlaced YUV420 data is fed into the DEI path. DEI is configured in deinterlacing mode. The deinterlaced input is scaled to 360x240 (YUYV422) and output via the WB0/WB1 path. Simultaneously the video is fed to the VIP0/1 through the transcode path and converted to 720x480 YUV420 progressive output.

- Please refer Common Steps for connecting CCS to TI816x, running gel file etc.
- Load *hdvpss_examples_m2mDeiScale.xem3* executable file found at *$(rel_folder)\build\bin\ti816x-evm\m3vpss\whole_program_debug*

to DSS M3 debug session

- Run the application
- The application will halt for the user to load the input frames and to select driver path. Using loadRaw command in script console of CCS, load 10 fields of

720 x 240 YUV420 semiplanar video to location mentioned in the console print. (Ignore "syntax error" if it appears during loading)

```
loadRaw(< Location >, 0, " < File Path > ", 32, false);
```

- Choose the required mode of driver from displayed options and enter in 'CCS console' window

```
Ex: For DEI WB1 single ouput driver, type '1' followed by 'enter' in
console window.
```

- User can save the outputs to a file using the saveraw command as printed from the console window.

```
saveRaw(0, < Location >, " < File Path > ", 432000, 32,
true);
saveRaw(0, < Location >, " < File Path > ", 1296000, 32,
true);
```

- Application will stop after processing 10 frames

DEI HQMQ-Mode1

This application shows an example of the DEIH-DEI Mode1 feature of the DEI M2M driver. It opens both the DEI instances, which provides dual output, in single channel per handle configuration mode. DEI takes even fields of the input channel and provides YUYV422 interleaved output from WB 0/1 path and also provides 360x240 (with 368 as pitch) YUV420 semi-planar output from VIP path. Odd fields of the the channel and YUYV422 interleaved output from the DEI as one field delayed input are fed to the DEIH. DEIH provides deinterlaced 360x240 (with 720 as pitch) YUYV422 interleaved output from DEIH writeback output and deinterlaced 720x480 YUV420 semi-planar output from VIP.

- Please refer Common Steps for connecting CCS to TI816x, running gel file etc.

Load hdvpss_examples_m2mDeiScale.xem3 executable file found at $(rel_folder)\build\bin\ti816x-evm\m3vpss\whole_program_debug

- Load *hdvpss_examples_m2mDeiHqMqMode1.xem3* executable file found at *$(rel_folder)\build\bin\ti816x-evm\m3vpss\whole_program_debug* to DSS M3 debug session
- Run the application after loading gets complete
- The application will halt for the user to load the input frames. Using loadRaw command in script console of CCS, load 20 fields of 720 x 240 YUV420 semiplanar video to location mentioned in the console print for one handle. (Ignore "syntax error" if it appears during loading)

```
loadRaw(< Location >, 0, " < File Path > ", 32, false);
```

- Once loading is completed. Press any key to continue. It prints output buffer addresses of DEI and VIP outputs. These addresses can be used to store output images into file using saveRaw command. saveRaw command dumps memory into a file. Use below command to save DEI SC_HQ output to a file

```
saveRaw(0, < Location >, " < File Path > ", 0x465000, 32,
true);
```

Use below command to save VIP0 output to a file

```
saveRaw(0, < Location >, " < File Path > ", 0x69780, 32,
true);
```

Use below command to save VIP1 output to a file

```
saveRaw(0, < Location >, " < File Path > ", 0x69780, 32,
true);
```

- Application will stop after processing 20 frames

UserGuideHdvpssTi814xDeiM2mDriver

TI814X/TI8107 Deinterlacer (DEI) Memory to Memory Driver

Introduction

This chapter describes the hardware overview, application software interfaces, typical application flow and sample application usage for DEI memory to memory driver.

The features and limitations of current driver implementation are listed in subsequent sections.

Important
The features supported or NOT supported in any release of the driver may vary from one HDVPSS driver release to another. See respective release notes for exact release specific details.

Features Supported

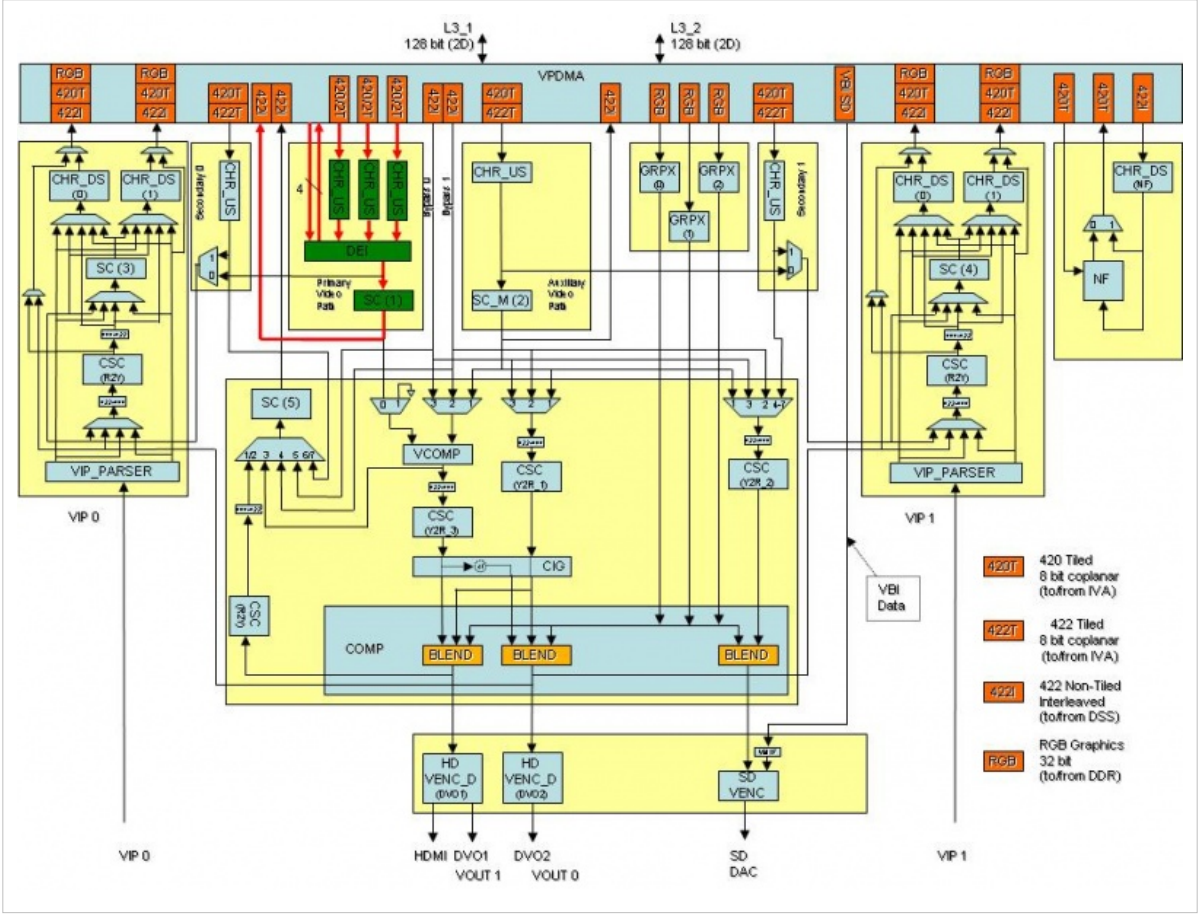
Features	Supported in TI814x	Supported in TI8107
Instances		
DEI_SC1_WB0 single scale path	YES	YES
DEI_SC3_VIP0 single scale path	YES	YES
DEI_SC1_SC3_WB0_VIP0 dual scale paths	YES	YES
SC2_WB1 single scale path	YES	YES
SC4_VIP1 single scale path	YES	YES
SC2_SC4_WB1_VIP1 dual scale paths	YES	YES
Input Formats		
YUV422 Interleaved	YES	YES
YUV420 Semi-Planar	YES	YES
YUV422 Semi-Planar	YES	YES

YUV422 Semi-Planar Tiled	NOT TESTED	NOT TESTED
YUV420 Semi-Planar Tiled	NOT TESTED	NOT TESTED
Output Formats		
YUV422 Interleaved on WB0	YES	YES
YUV422 Interleaved on VIP0	YES	YES
YUV420 Semi-Planar on VIP0	YES	YES
YUV422 Semi-Planar on VIP0	YES	NOT TESTED
YUV420 Semi-Planar Tiled on VIP0	NOT TESTED	NOT TESTED
YUV422 Semi-Planar Tiled on VIP0	NOT TESTED	NOT TESTED
RGB on VIP0	NO	NO
DEI Features		
DEI in deinterlacing mode	YES	YES
DEI in progressive bypass mode	YES	NOT TESTED
Compression enable/disable for previous field inputs	NO/NA	NO/NA
Line averaging and field averaging mode of DEI operation	YES	NOT TESTED
SC Features		
Optional scaling using SC3 and SC5	YES	YES
Scaling from 1/8x to 2048 maximum pixels in horizontal direction	YES	YES
Different types of scalar like poly phase and running average	YES	YES
Horizontal and vertical cropping of the image before scaling	YES	YES
User programmable scalar coefficients	YES	YES
Other Features		
Frame drop feature on WB0 and VIP0 outputs to enable load balancing	YES	YES
Multi-channel (up to `VPS_M2M_DEI_MAX_HANDLE_PER_INST` channels per instance)	YES	YES
Multi-handle (up to `VPS_M2M_DEI_MAX_HANDLE_PER_INST` channels per instance)	YES	YES
Error callbacks	YES	YES
Slice based scaling when DEI is in progressive bypass mode	NOT TESTED	NOT TESTED
Slice based scaling when DEI is in deinterlacing mode	NO	NO
Interlaced bypass mode	NO	NO
Runtime Configurations		
Input resolution change when DEI is in progressive bypass mode	YES	NOT TESTED
Input resolution change when DEI is in deinterlacing mode	YES	NOT TESTED
Output resolution change on WB0	YES	NOT TESTED
Output resolution change on VIP0	YES	YES
SC crop and config change on SC1/SC4	YES	YES
SC crop and config change on SC3/SC4	YES	YES
DEI reset	NOT TESTED	NOT TESTED

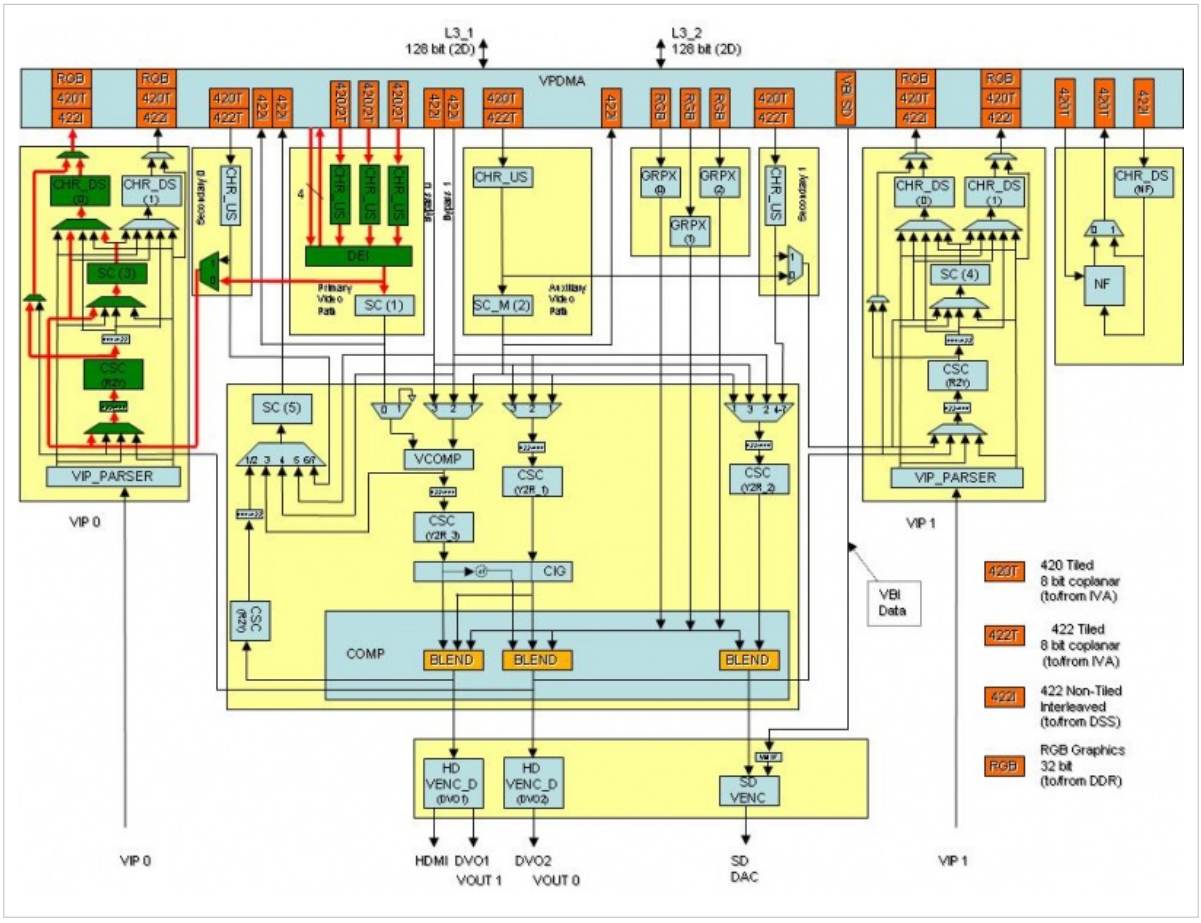
Hardware Overview

Below figures show the complete HDVPSS Hardware. The red bold lines in the figure shows the path on which the DEI memory to memory driver operates.

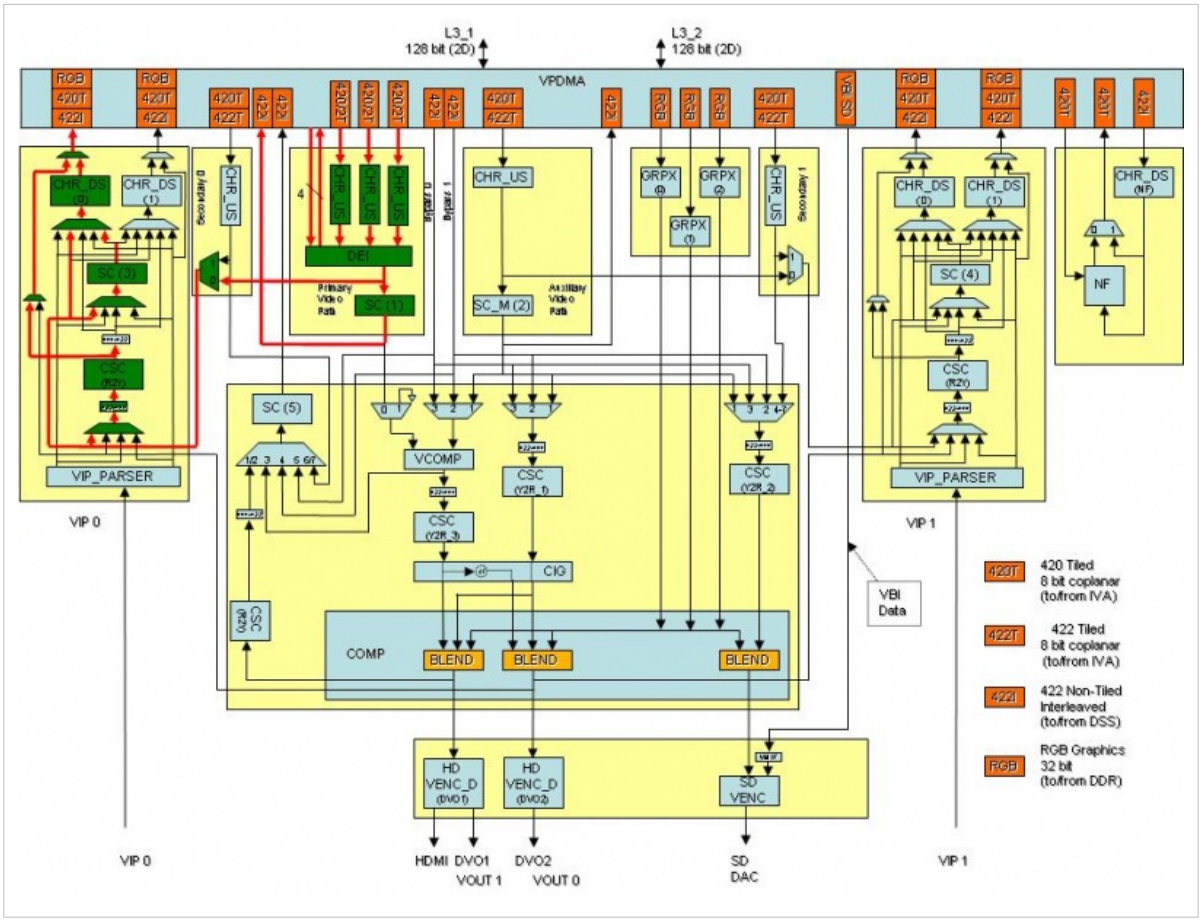
DEI-WB0 Single Output Path As shown in below figures, the DEI memory to memory driver takes in YUYV422/YUV420 interlaced/progressive input via the DEI path and provide single scaled output of the deinterlaced/bypassed output via writeback path 0



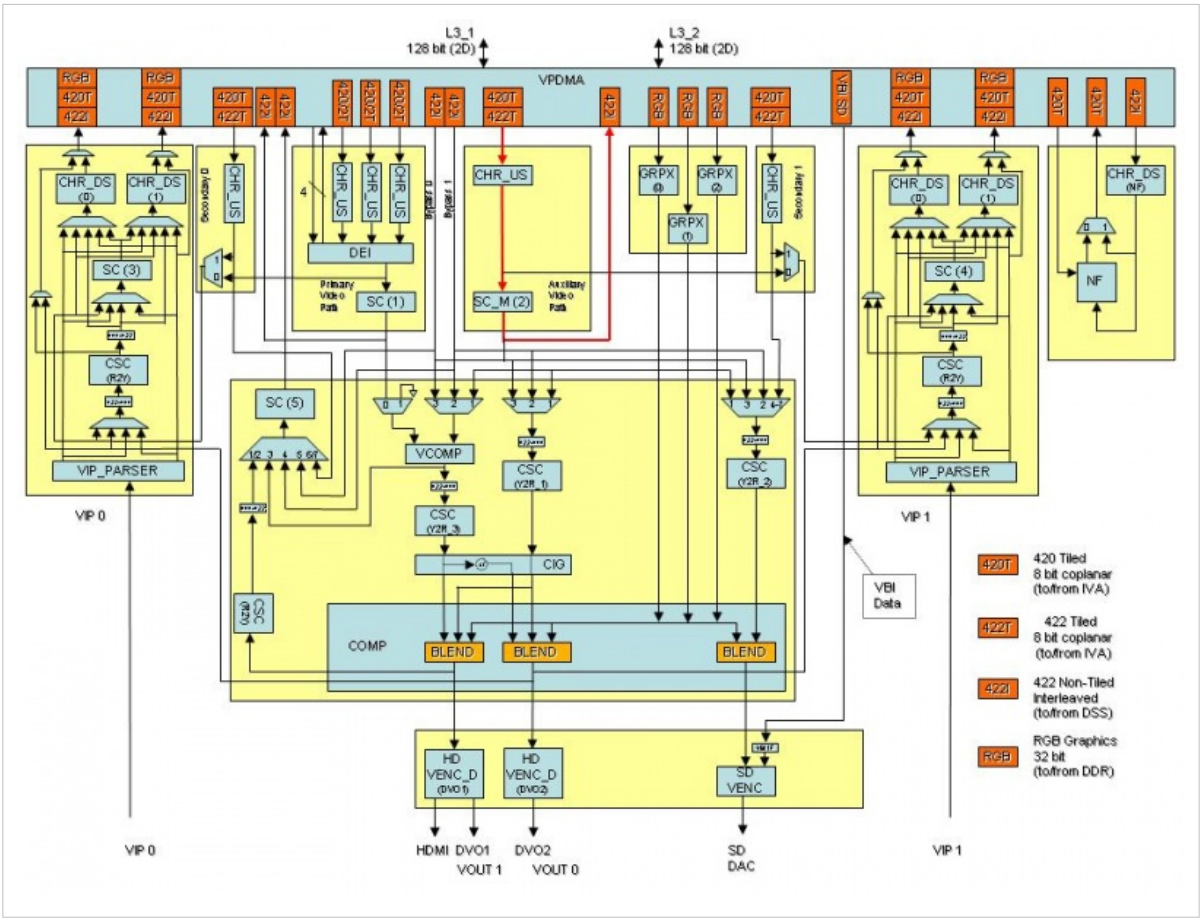
DEI-SC3-VIP0 Single Output Path As shown in below figures, the DEI memory to memory driver takes in YUYV422/YUV420 interlaced/progressive input via the DEI path and provide single scaled output of the deinterlaced/bypassed output via VIP path 0.



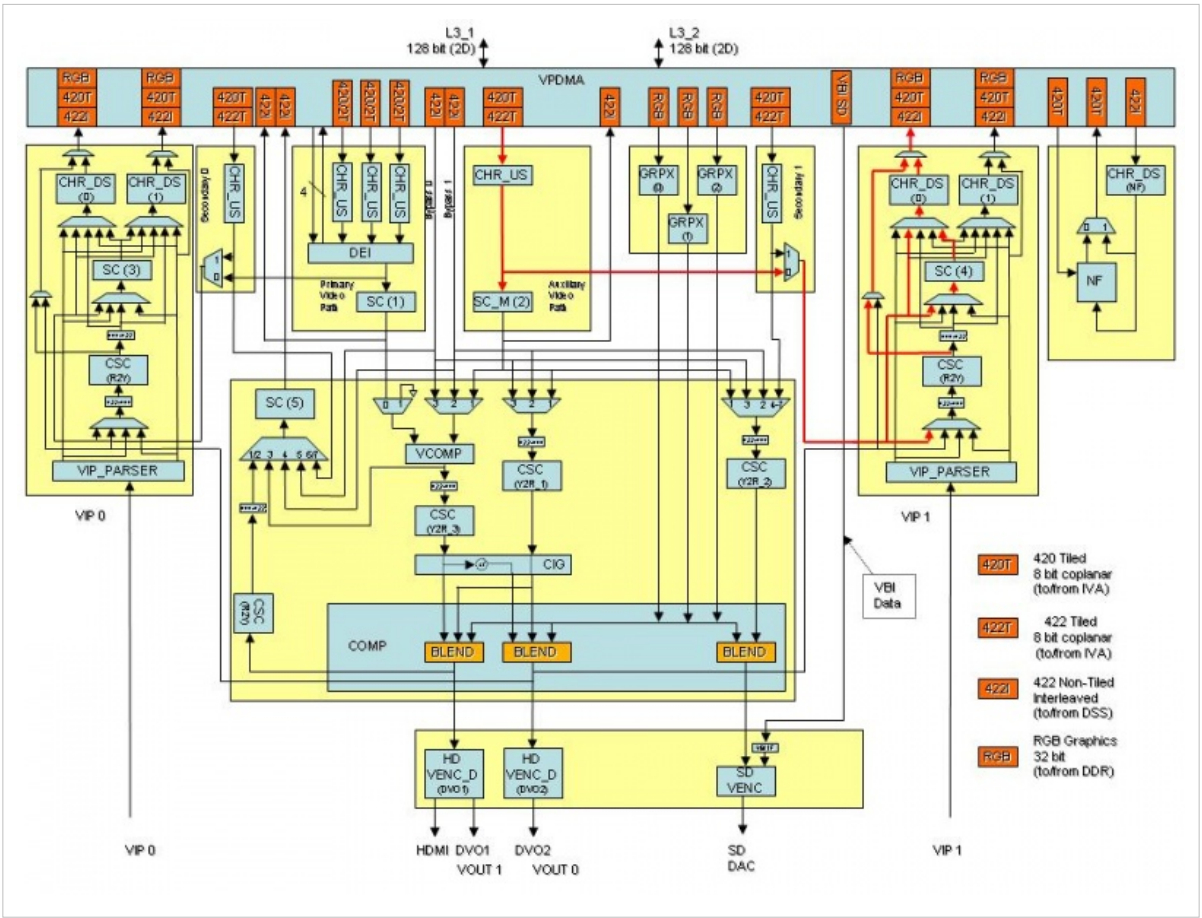
DEI-SC1-SC3-WB0 Dual Output Path As shown in below figures, the DEI memory to memory driver takes in YUYV422/YUV420 interlaced/progressive input via the DEI path and provide two scaled version of the deinterlaced/bypassed outputs - one via writeback path 0 and another via VIP 0.



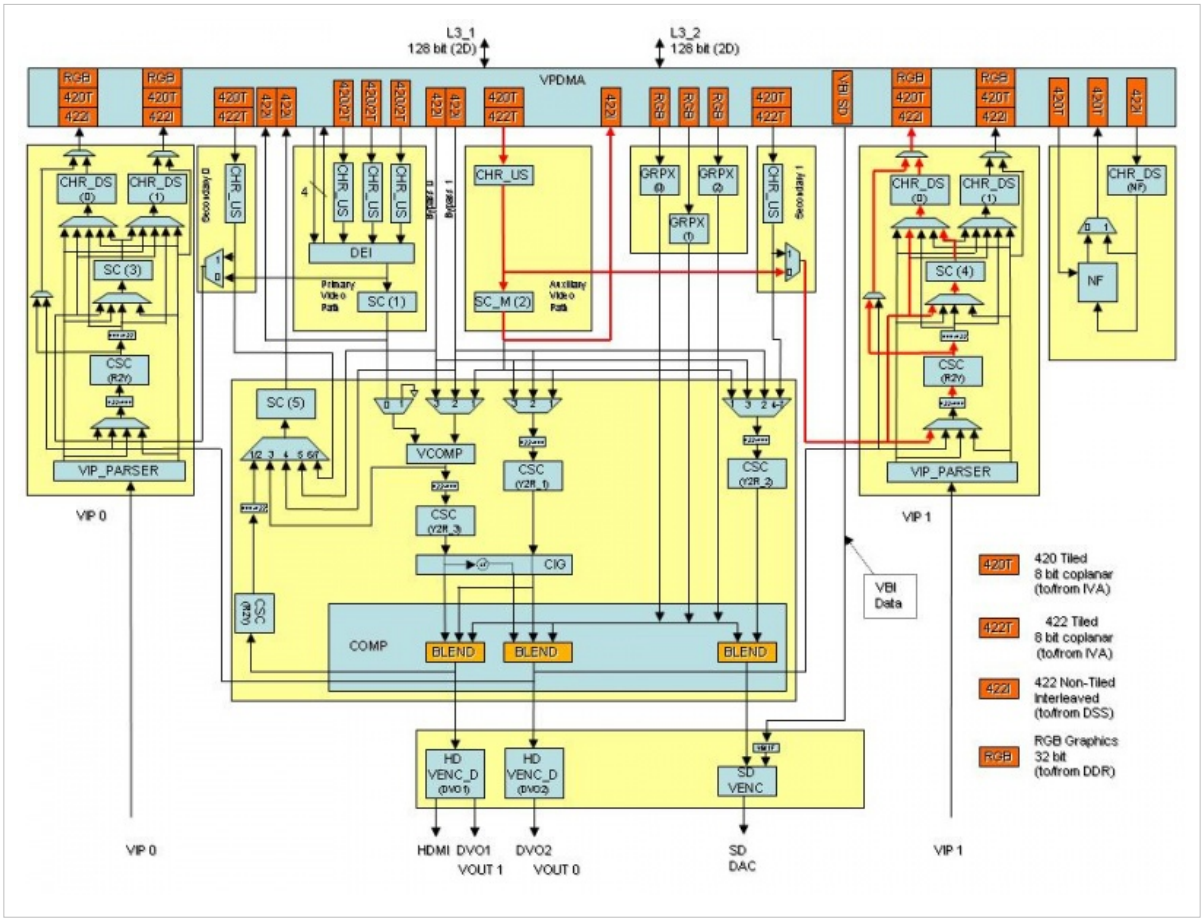
SC2-WB1 Single Output Path As shown in below figures, the SC memory to memory driver takes in YUYV422/YUV420 progressive input via the AUX path and provide single scaled output via writeback path 1 In this Instance of the driver we dont have DEI in its path, so only scaling is performed.



SC4-VIP1 Single Output Path As shown in below figures, the SC memory to memory driver takes in YUYV422/YUV420 progressive input via the AUX path and provide single scaled output via VIP path 1. In this Instance of the driver we dont have DEI in its path, so only scaling is performed.



SC2-SC4-WB1-VIP1 Dual Output Path As shown in below figures, the SC memory to memory driver takes in YUYV422/YUV420 progressive input via the AUX path and provide two scaled version of the outputs - one via writeback path 1 and another via VIP 1.



Overview

De-interlacing operation of a field requires two previous fields. This requirement is abstracted by the driver, the drivers holds back required input fields as context fields. The following expand on the behavior of the driver.

- Considering a single channel operation for DEI_SC1_WB0, following steps describe the operations performed on FVID2_ProcessList.inFrameList referred as inFrameList
- First FVID2_processFrames () assuming F1 was submitted to be de-interlaced, on completion of this API, FVID2_getProcessedFrames ()could be called to retrieve the output frame and input field. However, the input field would be held back by the driver (F1 in this case). i.e. The numFrames of inFrameList is set 0x1 and inFrameList->frames[0x0] = NULL. The output frame would be available, i.e. the numFrames of outFrameList is set to 0x01 and outFrameList->frames[0x0] will point to a valid frame. The applications are expected to check for valid frames, before performing further operations on the frame.

  inFrameList.numFrame = 0x01

  inFrameList.frames[0x0] = NULL

  outFrameList.numFrames = 0x01

  outFrameList.frames[0x0] = valid frame
- On second FVID2_processFrames () assuming F2 was submitted to be de-interlaced, the above step is repeated for field F2. i.e. F2 is also held back the driver
- On third FVID2_processFrames () assuming F3 was submitted to be de-interlaced, the above step is repeated for field F3. i.e. F3 is also held back the driver
- On fourth FVID2_processFrames () assuming F4 was submitted to be de-interlaced, on completion of this API, FVID2_getProcessedFrames ()could be called to retrieve the output frame and input field. The first

input field and fourth output frame is given back. i.e.

inFrameList.numFrame = 0x01

inFrameList.frames[0x0] = F1

outFrameList.numFrames = 0x01

outFrameList.frames[0x0] = valid frame

- On filth FVID2_processFrames () assuming F5 was submitted to be de-interlaced, the above step is performed with following output

  inFrameList.numFrame = 0x01

  inFrameList.frames[0x0] = F2

  outFrameList.numFrames = 0x01

  outFrameList.frames[0x0] = valid frame

- The driver would hold back previous N-1 and N-2 input fields as context buffers. This buffer could be retrieved using FIVD2_stop API.
- Multiple channel - The same procedures described for single channel applies. the numFrames of inFrameList and outFramesList defines the number of frames/fields that application should look for in the frames array.

  inFrameList.numFrame = 0x04

  inFrameList.frames[0x0] = CH1F1

  inFrameList.frames[0x1] = CH2F1

  inFrameList.frames[0x2] = NULL

  inFrameList.frames[0x3] = CH3F1

  outFrameList.numFrames = 0x04

  outFrameList.frames[0x0] = valid frame

  outFrameList.frames[0x1] = valid frame

  outFrameList.frames[0x2] = valid frame

  outFrameList.frames[0x3] = valid frame

Note that, in above example the driver has held back input field of channel 3, while releasing input fields of other channels. This would mean that channel 3 did not enough context buffers.

Important

Applications should take into account that any input field could be held back by the driver, an NULL check on field should be performed before using it.

Software Application Interfaces

The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: Here driver is used to submit the frames for processing and getting the processed frames from the driver.
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is de-initialized

The subsequent sections describe each phase in detail.

Note
Details of the structure, enumerations and #defines mentioned in the section can be found in HDVPSS API Guide

System Init Phase

DEI M2m driver initialization happens as part of overall HDVPSS system init. This API must be the the first API call before making any other FVID2 calls. Below section lists the APIs which are part of the System Init phase.

FVID2 Init

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

```
    /* Init FVID2 and VPS */
    retVal = FVID2_init(NULL);
    if (FVID2_SOK != retVal)
    {
        System_printf("FVID2 Init failed\n");
    }
```

Create Phase

In this phase user application opens or creates a driver instance. Each instance of the driver supports VPS_M2M_DEI_MAX_HANDLE_PER_INST (defined in vps_m2mDei.h) handles creation. Operation commands from the different handles of the same instance will be serialized by the driver and will be served by the singe instance of the hardware. Below sections lists the API interfaces to be used in the create phase. Create phase allows the application to do the configuration either through control commands exposed by driver or through the parameters passed with the driver create API.

For VPS_M2M_INST_AUX_SC2_WB1, VPS_M2M_INST_AUX_SC4_VIP1 and VPS_M2M_INST_AUX_SC2_SC4_WB1_VIP1 instances of the driver dei related params in the createargs should be NULL.

The above three instances will use the existing DEI driver interface but has no DEI in their path, so dei related parameters should be passed as NULL from the application during create time. deiCfg param which is member of Vps_M2mDeiChParams structure should be assigned NULL for new instances. deiRtCfg param of Vps_M2mDeiRtParams structure should also be passed as NULL, if runtime parameters are used.

FVID2 Create

This API is used to open the driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

```
FVID2_Handle FVID2_create(UInt32 drvId,
                          UInt32 instanceId,
                          Ptr createArgs,
                          Ptr createStatusArgs,
                          const FVID2_CbParams *cbParams);
```

drvId - `FVID2_VPS_M2M_DEI_DRV` to open the driver.

instanceId - `VPS_M2M_INST_MAIN_DEI_SC1_SC3_WB0_VIP0` macro to open DEI_SC1_SC3_WB0_VIP0 dual scale memory driver.

`VPS_M2M_INST_MAIN_DEI_SC1_WB0` macro to open DEI_SC1_WB0 single scale memory driver.

`VPS_M2M_INST_MAIN_DEI_SC3_VIP0` macro to open DEI_SC3_VIP0 single scale memory driver.

`VPS_M2M_INST_AUX_SC2_SC4_WB1_VIP1` macro to open SC2_SC4_WB1_VIP1 dual scale memory driver.

`VPS_M2M_INST_AUX_SC2_WB1` macro to open SC2_WB1 single scale memory driver.

`VPS_M2M_INST_AUX_SC4_VIP1` macro to open SC4_VIP1 single scale memory driver.

createArgs - Pointer to `Vps_M2mDeiCreateParams` structure containing valid create params. This parameter should not be NULL.

createStatusArgs - Pointer to `Vps_M2mDeiCreateStatus` structure containing the return value of create function and other driver information. This parameter should not be NULL.

cbParams - Pointer to `FVID2_CbParams` structure containing FVID2 callback parameters. This parameter should not be NULL.

```
    FVID2_Handle            fvidHandle;
    FVID2_CbParams          cbParams;
    Vps_M2mDeiChParams      chPrms;
    Vps_M2mDeiCreateParams  createParams;
    Vps_M2mDeiCreateStatus  createStatus;

    /* Init create params */
    createParams.mode = VPS_M2M_CONFIG_PER_CHANNEL;
    createParams.numCh = 1u;
    createParams.deiHqCtxMode = VPS_DEIHQ_CTXMODE_DRIVER_ALL;
    createParams.chParams = &chPrms;
    createParams.isVipScReq = TRUE;

    /* Init callback parameters */
    cbParams.cbFxn = App_m2mDeiAppCbFxn;
    cbParams.errCbFxn = App_m2mDeiAppErrCbFxn;
    cbParams.errList = &errProcessList;
    cbParams.appData = NULL;
    cbParams.reserved = NULL;

    /* Open the driver */
    fvidHandle = FVID2_create(
```

```
                    FVID2_VPS_M2M_DEI_DRV,
                    VPS_M2M_INST_MAIN_DEI_SC3_VIP0,
                    &createParams,
                    &createStatus,
                    &cbParams);
    if (NULL == fvidHandle)
    {
        System_printf("Create failed!!\n");
    }
```

FVID2 Control - Set Scalar Coefficient

This is used to issue a control command to the driver. `IOCTL_VPS_SET_COEFFS` ioctl is used to set the scalar coefficients. This is a blocking call.

Important
This API should not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SET_COEFFS` ioctl.

cmdArgs - Pointer to `Vps_ScCoeffParams` structure containing valid scaling coefficient. This parameter should not be NULL. To set the scalar coefficient for DEI scalar, `scalarId` should be set to `VPS_M2M_DEI_SCALAR_ID_DEI_SC` and for VIP scalar `scalarId` should be set to `VPS_M2M_DEI_SCALAR_ID_VIP_SC`.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Get DEI Context Information

When DEI is in de-interlacing mode, DEI requires previous fields and motion vectors. Buffers for storing these context information must be allocated by the application and provided to the driver before starting M2M operation. `IOCTL_VPS_GET_DEI_CTX_INFO` ioctl is used to get the number of internal buffers to be allocated and their sizes. Application should get this information from the driver and allocate these buffers and provide the buffers to the driver before issuing any request. Once these buffers are given to the driver, application should not modify these buffers. This should be done for each and every channel. This is a blocking call. This IOCTL is not supported for VPS_M2M_INST_AUX_SC2_WB1, VPS_M2M_INST_AUX_SC4_VIP1 and VPS_M2M_INST_AUX_SC2_SC4_WB1_VIP1 instances.

Important
This API should not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_GET_DEI_CTX_INFO` ioctl.

cmdArgs - Pointer to `Vps_DeiCtxInfo` structure where the DEI context information will be filled by driver. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Set DEI Context Buffers

`IOCTL_VPS_SET_DEI_CTX_BUF` ioctl is used to set the DEI context buffers for a channel before providing any request to the driver. This is a blocking call. This IOCTL is not supported for VPS_M2M_INST_AUX_SC2_WB1, VPS_M2M_INST_AUX_SC4_VIP1 and VPS_M2M_INST_AUX_SC2_SC4_WB1_VIP1 instances.

Important
This API should not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SET_DEI_CTX_BUF` ioctl.

cmdArgs - Pointer to `Vps_DeiCtxBuf` structure valid buffer pointers as requested by the driver for a particular DEI mode of operation. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Get DEI Context Buffers

`IOCTL_VPS_GET_DEI_CTX_BUF` ioctl is used to get the DEI context buffers for a channel from the driver. Once the DEI context buffer is returned to the application, no more request should be provided to the driver. This is a blocking call. This IOCTL is not supported for VPS_M2M_INST_AUX_SC2_WB1, VPS_M2M_INST_AUX_SC4_VIP1 and VPS_M2M_INST_AUX_SC2_SC4_WB1_VIP1 instances.

Important
This API should not be called when there are any pending request with the driver.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_GET_DEI_CTX_BUF` ioctl.

cmdArgs - Pointer to `Vps_DeiCtxBuf` structure where the driver returns back the DEI context buffer to the application. This parameter should not be NULL.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

Run Phase

M2m drivers are non-streaming drivers. This phase is used to submit the requests for processing and getting the processes request back.

Start

NA

Stop

The driver would retained 3 fields, as context fields. Once application has completed all the de-interlacing operation. This command could be used to retrieve the context fields.

- Should only be used when de-interlacing, i.e. should not be used in bypass mode
- Normally when applications are ready to close, this control command is expected to be used.
- Is a blocking call

```
Int32 FVID2_stop(FVID2_Handle handle,
                 Ptr cmdArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmdArgs - Not used currently. This parameter should be set to NULL.

FVID2 Process Frames

This API is used to submit video buffers to the driver for processing operation. This is a non-blocking call and should be called from task context. Once the buffer is queued the application loses ownership of the buffer and is not suppose to modify or use the buffer.

```
Int32 FVID2_processFrames(FVID2_Handle handle,
                          FVID2_ProcessList *processList);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

processList - Pointer to `FVID2_ProcessList` structure containing the pointer to the FVID2 frames/framelist. This parameter should not be NULL.

FVID2 Get Processed Frames

This API is used by the application to get ownership of the processed video buffer from the memory driver. This is a non-blocking call and could be called from task or ISR context.

```
Int32 FVID2_getProcessedFrames(FVID2_Handle handle,
                               FVID2_ProcessList *processList,
                               UInt32 timeout);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

processList - Pointer to `FVID2_ProcessList` structure where the driver will copy the processed FVID2 frames/framelist. This parameter should not be NULL.

timeout - Not used currently as only non-blocking queue/dequeue operation is supported. This parameter should be set to FVID2_TIMEOUT_NONE.

Delete Phase

In this phase FVID2 delete API is called to close the driver handle. Hardware resources are freed once all the handles of the particular instance are freed. Handle can be opened again, once close, with different configuration.

FVID2 Delete

This API is used to close the memory driver. This is a blocking call and returns after closing the handle.

```
Int32 FVID2_delete(FVID2_Handle handle, Ptr deleteArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

deleteArgs - Not used currently. This parameter should be set to NULL.

System De-Init Phase

FVID2 de-Init

Drivers gets de-initializes as a part of HDVPSS sub-system de-Initialization. Here all resources acquired during system initialization are free'ed. Make sure all driver instances and handles are deleted before calling this API. Typically this is done during system shutdown.

```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

Sample Application

DEI Single and Dual Scale

This example illustrates the three paths supported by DEI memory to memory driver involving single and dual scaling outputs as listed below. Driver path can be selected using user input through console.

- DEI Single Scale: DEI_SC1_WB0
- DEI-VIP0 Single Scale: DEI_SC3_VIP0
- DEI-VIP0 Dual Scale: DEI_SC1_SC3_WB0_VIP0

DEI Single Scale example features: 720x240 interlaced YUV420 data is fed into the DEI path. DEI is configured in deinterlacing mode. The deinterlaced input is scaled to 360x240 (YUV422) and output via the WB0 path.

DEI-VIP0 Single Scale example features: 720x240 interlaced YUV420 data is fed into the DEI path. DEI is configured in deinterlacing mode. The deinterlaced input is is fed to the VIP0 through the transcode path and converted to 720x480 YUV420 progressive output.

DEI-VIP0 Dual Scale example features: 720x240 interlaced YUV420 data is fed into the DEI path. DEI is configured in deinterlacing mode. The deinterlaced input is scaled to 360x240 (YUYV422) and output via the WB0 path. Simultaneously the video is fed to the VIP0 through the transcode path and converted to 720x480 YUV420 progressive output.

- Please refer Common Steps for connecting CCS to TI814x/TI8107, running gel file etc.
- Load *hdvpss_examples_m2mDeiScale.xem3* executable file found at *$(rel_folder)\build\bin\ti8107-evm\m3vpss\whole_program_debug*

to DSS M3 debug session

- Run the application
- The application will halt for the user to load the input frames and to select driver path. Using loadRaw command in script console of CCS, load 10 fields of

720 x 240 YUV420 semiplanar video to location mentioned in the console print. (Ignore "syntax error" if it appears during loading)

```
loadRaw(< Location >, 0, " < File Path > ", 32, false);
```

- Choose the required mode of driver from displayed options and enter in 'CCS console' window

```
Ex: For DEI WB1 single ouput driver, type '1' followed by 'enter' in
console window.
```

- User can save the outputs to a file using the saveraw command as printed from the console window.

```
saveRaw(0, < Location >, " < File Path > ", 432000, 32,
true);
saveRaw(0, < Location >, " < File Path > ", 1296000, 32,
true);
```

- Application will stop after processing 10 frames

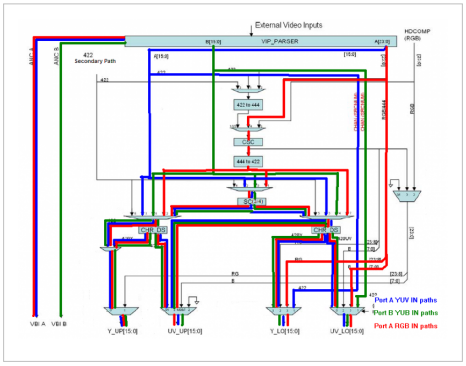
UserGuideHdvpssCaptureDriver

Introduction

VIP capture driver makes use of VIP hardware block in HDVPSS to capture data from external video source like video decoders (example, TVP5158, TVP7002). The video data is captured from the external video source by the VIP Parser sub-block in the VIP block. The VIP Parser then sends the captured data for further processing in the VIP block which can include colour space conversion, scaling, chroma down sampling and finally writes the video data to external DDR memory.

The data paths supported by the current driver implementation are shown in the below figure:

Important
Only multichannel capture is tested on TI8107 platform



Features supported

Features	**Supported in TI816x**	**Supported in TI814x**	**Supported in TI8107**
Input Video Source Formats			
YUV422 8-bit embedded sync mode	YES	YES	YES
YUV422 16-bit embedded sync mode	YES	YES	NOT TESTED
RGB 24-bit embedded sync mode	YES	NOT TESTED	NOT TESTED
YUV422 8-bit discrete sync mode	NOT TESTED	NOT TESTED	NOT TESTED
YUV422 16-bit discrete sync mode	YES	YES	NOT TESTED
RGB 24-bit discrete sync mode	YES	NOT TESTED	NOT TESTED
YUV422 8-bit 2x/4x pixel multiplxed mode	YES	YES	YES
YUV422 8-bit 4x line multiplexed mode	YES	YES	YES
YUV422 8-bit 4x split-line multiplexed mode	NO	NO	NO
YUV444 24-bit embedded/discrete sync mode	NO	NO	NO
Output Video formats			
YUV422 YUYV interleaved format	YES	YES	YES
YUV420 Semi-planer format	YES	YES	NOT TESTED
RGB 24-bit interleaved format	YES	YES	NOT TESTED
YUV422 Semi-planer format	YES	NO	NOT TESTED
In-line video processing features			
Color space conversion	YES	YES	NOT TESTED
Down-Scaling	YES	YES	NOT TESTED
Chroma-down sampling	YES	YES	NOT TESTED
Other features			
Multi-instance (VIP0, VIP1), multi-port capture (Port A, Port B), with ability to configure each instance, port independently	YES	YES	YES
Interlaced as well as progressive capture	YES	YES	YES
Non-multiplexed capture upto Dual 1080P60 (1920x1080) resolution using 2 VIP ports (VIP0/A, VIP1/A)	YES	YES	NOT TESTED
Multi-channel - upto 16CH D1 (NTSC/PAL) using 4 VIP ports (VIP0/A, VIP0/B, VIP1/A, VIP1/B)	YES	YES For 8 Channels on VIP0 Port A/B	YES For 4 Channels on VIP0 Port A
Multi-channel - upto 32CH Half-D1/CIF (NTSC/PAL) using 4 VIP ports (VIP0/A, VIP0/B, VIP1/A, VIP1/B)	NOT TESTED	NOT TESTED	NOT TESTED
Per frame info to user like - field ID, captured frame width x height, timestamp, logical channel ID	YES	YES	YES
Frame-rate depends on external video source, no limitation in driver as such	YES	YES	YES
For RGB input, optional color space conversion to YUV is supported	YES	NOT TESTED	NOT TESTED
For RGB input with color space conversion to YUV enabled, optional scaling is supported	YES	NOT TESTED	NOT TESTED
For RGB input with color space conversion to YUV enabled, optional scaling is supported	YES	NOT TESTED	NOT TESTED

For YUV input, optional color space conversion to RGB is supported	YES	NOT TESTED	NOT TESTED
For YUV input, optional scaling is supported	YES	YES	NOT TESTED
For YUV input, optional chroma downsampling is supported	YES	YES	NOT TESTED
Per channel frame-dropping. Example, for a 60fps video source, 30fps, 15fps, 7fps capture	YES	YES	NOT TESTED
Ability to change scalar parameters while capture is running	YES	NOT TESTED	NOT TESTED
Single source (RGB 24-bit or YUV422 8/16-bit), dual output (RGB 24-bit and/or YUV422 and/or YUV420) support. See table below for support combinations	YES	YES	NOT TESTED
Raw VBI capture for single/multi channel modes	YES	NOT TESTED	NOT TESTED
Non-blocking FVID2 queue, dequeue API support	YES	YES	YES
VIP resource management is supported for VIP capture driver	YES	YES	YES
Create time path allocation is supported i.e. when a capture driver is opened for a particular input to output combination, the driver will select a path that is not used by any other VIP capture driver or M2M driver that uses VIP in its path. Path allocation is possible only at create time. Run-time path switching is not possible	YES	YES	YES
Possible to configure VIP port for different video input source properties like Hsync polarity, Vsync polarity, PCLK polarity	YES	YES	NOT TESTED
Possible to configure custom scaling co-effs during create time	YES	NOT TESTED	NOT TESTED
Tiler memory support when output type is YUV420 semi-planer	YES	NOT TESTED	NOT TESTED
Sub-frame based capture	NO	NO	NO

- In the table above,
  Support = YES, means feature has been tested with current driver on current platform board/EVM.
  Support = NO, means feature is not supported in current driver and using it will give unpredictable results.
  Feature is planned to be supported in future releases.
  Support = NOT TESTED, means feature is present in driver but has NOT been tested on due to current platform board/EVM limitations AND/OR is planned to be tested in subsequent releases.

Input to Output Combinations support

Input Format	Output format - 0	Output format - 1	Support in TI816x	Supoprt in TI814x

YUV422 8/16-bit embedded sync mode	YUV422 YUYV interleaved format (optionally scaled)	NONE	YES	YES
	YUV420 Semi-planer format (optionally scaled)	NONE	YES	YES
	RGB 24-bit interleaved format (via CSC)	NONE	YES	NOT TESTED
	YUV422 Semi-planer format (optionally scaled)	NONE	YES	NO
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	YES	YES
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format (via CSC)	YES	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	YES	YES
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	YES	NO
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	YES	YES
	YUV420 Semi-planer format (optionally scaled)	RGB 24-bit interleaved format (via CSC)	YES	NOT TESTED
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	YES	NO
YUV444 24-bit embedded/discrete sync mode	NA	NA	NO	NO
YUV422 8/16-bit embedded sync mode - MULTI-CH modes - pixel mux, line mux	YUV422 YUYV interleaved format (SCALING, CHR_DS, CSC NOT SUPPORTED in MULTI-CH mdoes)	NONE (Dual output not supported in MULTI-CH modes)	YES	YES

RGB 24-bit discrete sync mode (CSC is used when output format is YUV)	YUV422 YUYV interleaved format (optionally scaled)	NONE	YES	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	NONE	YES	NOT TESTED
	RGB 24-bit interleaved format	NONE	YES	NOT TESTED
	YUV422 Semi-planer format (optionally scaled)	NONE	YES	NO
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	YES	NOT TESTED
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format	YES	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	YES	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	YES	NO
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	YES	NOT TESTED
	RGB 24-bit interleaved format	YUV420 Semi-planer format (optionally scaled)	YES	NOT TESTED
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	YES	NO

RGB 16-bit discrete sync mode (CSC is used when output format is YUV)	YUV422 YUYV interleaved format (optionally scaled)	NONE	YES	YES
	YUV420 Semi-planer format (optionally scaled)	NONE	YES	YES
	RGB 24-bit interleaved format	NONE	YES	YES
	YUV422 Semi-planer format (optionally scaled)	NONE	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NOT TESTED	NOT TESTED
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	NOT TESTED	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	RGB 24-bit interleaved format	YES	YES
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NOT TESTED	NOT TESTED

RGB 8-bit discrete sync mode (CSC is used when output format is YUV)	YUV422 YUYV interleaved format (optionally scaled)	NONE	NOT TESTED	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	NONE	NOT TESTED	NOT TESTED
	RGB 24-bit interleaved format	NONE	NOT TESTED	NOT TESTED
	YUV422 Semi-planer format (optionally scaled)	NONE	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NOT TESTED	NOT TESTED
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	NOT TESTED	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED	NOT TESTED
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NOT TESTED	NOT TESTED

RGB 24-bit embedded sync mode (CSC is used when output format is YUV)	YUV422 YUYV interleaved format (optionally scaled)	NONE	NOT TESTED	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	NONE	NOT TESTED	NOT TESTED
	RGB 24-bit interleaved format	NONE	YES	NOT TESTED
	YUV422 Semi-planer format (optionally scaled)	NONE	YES	NO
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	NOT TESTED	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	YES	NO
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	NOT TESTED	NOT TESTED
	RGB 24-bit interleaved format	YUV420 Semi-planer format (optionally scaled)	YES	NOT TESTED
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	YES	NO

- In the table above,
  Support = YES, means feature has been tested with current driver on current platform board/EVM.
  Support = NO, means feature is not supported in current driver and using it will give unpredictable results.
  Feature is planned to be supported in future releases.
  Support = NOT TESTED, means feature is present in driver but has NOT been tested on due to current platform board/EVM limitations AND/OR is planned to be tested in subsequent releases.

Limitations/Issues

- There are limiations in capture driver for features like video source cable disconnect/connect, discrete sync mode, VIP parser overflow, Chroma downsampling. These limiations are related to Si issues. Please refer to Si Errata document to get latest update and workarounds for these issues.

Software Application Interfaces

The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: Here the driver is used to capture, process and release frames continuously
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is de-initialized

The subsequent sections describe each phase in detail.

System Init Phase

The VIP capture driver sub-system initialization happens as part of overall HDVPSS system init. Below code shows the FVID2 API used to initialize the overall HDVPSS subsystem. This API must be the the first API call before making any other FVID2 calls.

An example is shown below. Also shown in the example there is a FVID2 create API call to create the global VIP capture handle. This handle, as shown later, can be used to queue, dequeue frames from all active VIP ports.

```
#include "ti/psp/vps/vps_capture.h"

FVID2_Handle fvidHandleVipAll;

Int32 mySysInit()
{
  Int32 status;
  FVID2_CbParams cbPrm;

  /* FVID2 system init */
  status = FVID2_init(NULL);

  assert(status==0);

  /* must be NULL for VPS_CAPT_INST_VIP_ALL */
  memset(& cbPrm, 0, sizeof(cbPrm));

  /* Create global VIP capture handle, used for dequeue,
      queue from all active captures */
  fvidHandleVipAll = FVID2_create(
                       FVID2_VPS_CAPT_VIP_DRV,
                       VPS_CAPT_INST_VIP_ALL,
                       NULL, /* must be NULL for VPS_CAPT_INST_VIP_ALL */
                       NULL, /* must be NULL for VPS_CAPT_INST_VIP_ALL */
                       & cbPrm
                     );

  assert(fvidHandleVipAll!=NULL);

  return status;
}
```

Internally the following happens when VIP capture initialization is done via FVID2 init:

- Hardware resources like interrupts, hardware lists are allocated
- Driver name is registered with FVID2 sub-system
- In addition to doing FVID2_init(), some platform specific init may also be required, like initializing I2C's, video PLL's etc. Refer to sample code directly for sample platform specific init.

Create Phase

In this phase user application opens or creates a driver instance. Up to VPS_CAPT_INST_MAX (defined in vps_capture.h) driver instances can be opened by a user. A driver instance is associated with one or more VIP parser ports depending on whether the operation mode is 8-bit or 16-bit or 24-bit.

User can pass a number of parameters during create which controls the mode in which the driver instance gets created, example single channel or multi-channel mode. Refer to VIP Capture section in HDVPSS API Guide for detailed list of create time parameters.

Driver Instance to hardware port mapping for different bus-widths

The mapping of driver instance to VIP parser ports in HDVPSS is shown below:

Driver Instance	8-bit interface	16-bit interface	24-bit interface
VPS_CAPT_INST_VIP0_PORTA	VIP0 PortA	VIP0 PortA	VIP0 PortA
VPS_CAPT_INST_VIP0_PORTB	VIP0 PortB	NOT USED	NOT USED
VPS_CAPT_INST_VIP1_PORTA	VIP1 Port A	VIP1 Port A	NOT USED (TI816x) VIP1 PortA (TI 814x)
VPS_CAPT_INST_VIP1_PORTB	VIP1 Port B	NOT USED	NOT USED

Output streams

A maximum of two output streams (excluding VBI capture) are possible from the capture driver in non-multiplexed modes of capture. Data from each stream can be independently queued/dequeued when capture data streaming is enabled using FVID2_start().

Refer to table in previous section for valid supported input / output combinations for different input source formats.

Example of streams are:

- Single source dual format capture - YUV420 capture (stream 0) + RGB capture (stream 1)
- Ancillary data capture - YUV422 capture (stream 0) + VBI capture (stream 1)

NOTE

Channel is different from stream in the sense that channel is associated with a distinct input source. For different output streams the input source (or channel) is the same, however the final output format - data format (RGB, YUV422, YUV420), or resolution, or data type (VBI, active data) - is different for each output stream. Thus when capturing 4CH D1 through one VIP port, number of valid channels will be four and output streams would be one (YUV422 format). However when capturing single channel 24-bit RGB, number of output streams can be three - YUV420 (stream 0), RGB 24-bit (stream 1), Ancillary data (stream 3).

NOTE

If FVID2_DF_YUV422SP_UV is used as output format, it must be the first output format (output format at the index 0 in outStreamInfo of Vps_CaptCreateParams).

FVID2 create

The FVID2 create call that is used to create a capture instance is shown below. The example shows FVID2 create used to create two output streams of RGB and scaled YUV420 output from single 24-bit RGB input source.

Important

UInt32 Vps_CaptCreateParams.channelNumMap[VPS_CAPT_STREAM_ID_MAX] [VPS_CAPT_CH_PER_PORT_MAX] is the channel number assigned by user for every channel in every output stream for a given instance during FVID2 create.

This channel number will be returned with the FVID2 frame when it is captured by the driver.

Channel number is used internally by the driver to identify the driver instance, output stream, input source a frame belongs to.
Channel number can also be used by the user for the same purpose.

Further internally driver needs that the channel number across all VIP capture instances to be unique. Its upto the user to ensure that this condition is met.

A utility API, UInt32 Vps_captMakeChannelNum(UInt32 instId, UInt32 streamId, UInt32 chId) , is provided to help user generate such system unique channel numbers.
However user is free to use their own mechanism for generating system unique channel numbers.

A channel number MUST be between 0 and 255.

```
#include "ti/psp/vps/vps_capture.h"
#include "ti/psp/vps/common/vpsutils_mem.h"

#define CAPTURE_APP_FRAMES_PER_CH     (6) // MUST be <=
VPS_CAPT_FRAME_QUE_LEN_PER_CH_MAX

typedef struct {

  UInt32 instId;
  Vps_CaptCreateParams  createArgs;
  Vps_CaptCreateStatus  createStatus;
  FVID2_CbParams        cbPrm;
  FVID2_Handle          fvidHandle;

  FVID2_Frame   frames[
    VPS_CAPT_STREAM_ID_MAX*
    VPS_CAPT_CH_PER_PORT_MAX*
    CAPTURE_APP_FRAMES_PER_CH
    ];

  Vps_CaptRtParams  rtParams[
    VPS_CAPT_STREAM_ID_MAX*
    VPS_CAPT_CH_PER_PORT_MAX*
    CAPTURE_APP_FRAMES_PER_CH
    ];

} CaptureApp_DrvObj;
```

```
Int32 CaptureApp_create(CaptureApp_DrvObj *pDrvObj)
{
  UInt16 chId, streamId, instId = VPS_CAPT_INST_VIP0_PORTA;

  pDrvObj->instId = instId;

  memset(& pDrvObj->createArgs, 0,
sizeof(Vps_CaptCreateParams));

  pDrvObj->createArgs.videoCaptureMode
         =
VPS_CAPT_VIDEO_CAPTURE_MODE_SINGLE_CH_NON_MUX_EMBEDDED_SYNC;

  pDrvObj->createArgs.videoIfMode       =
VPS_CAPT_VIDEO_IF_MODE_24BIT;
  pDrvObj->createArgs.inDataFormat      = FVID2_DF_RGB24_888;

  pDrvObj->createArgs.enablePeriodicCallback = FALSE;

  pDrvObj->createArgs.numCh             = 1;
  pDrvObj->createArgs.numStream         = 2;

  pDrvObj->createArgs.outStreamInfo[0].dataFormat =
FVID2_DF_YUV420SP_UV;
  pDrvObj->createArgs.outStreamInfo[0].memType  =
VPS_VPDMA_MT_NONTILEDMEM;
  pDrvObj->createArgs.outStreamInfo[0].pitch[0] = pitch of Y plane
data, MUST be multiple of 16bytes;
  pDrvObj->createArgs.outStreamInfo[0].pitch[1] = pitch of C plane
data, MUST be multiple of 16bytes;
  pDrvObj->createArgs.outStreamInfo[0].scEnable = TRUE;
  pDrvObj->createArgs.outStreamInfo[0].subFrameModeEnable = FALSE;
  pDrvObj->createArgs.outStreamInfo[0].maxOutHeight =
VPS_CAPT_MAX_OUT_HEIGHT_1080_LINES;


  pDrvObj->createArgs.outStreamInfo[1].dataFormat =
FVID2_DF_RGB24_888;
  pDrvObj->createArgs.outStreamInfo[1].memType  =
VPS_VPDMA_MT_NONTILEDMEM;
  pDrvObj->createArgs.outStreamInfo[1].pitch[0] = pitch of RGB
24-bit data, MUST be multiple of 16bytes;
  pDrvObj->createArgs.outStreamInfo[1].scEnable = FALSE;
  pDrvObj->createArgs.outStreamInfo[1].sliceModeEnable = FALSE;
  pDrvObj->createArgs.outStreamInfo[1].subFrameModeEnable = FALSE;
  pDrvObj->createArgs.outStreamInfo[1].maxOutHeight =
VPS_CAPT_MAX_OUT_HEIGHT_1080_LINES;
```

```
  pDrvObj->createArgs.scParams.inWidth = input source width in
pixels;
  pDrvObj->createArgs.scParams.inHeight= input source height in
lines;

  pDrvObj->createArgs.scParams.outWidth
        = pDrvObj->createArgs.scParams.inWidth/2;

  pDrvObj->createArgs.scParams.outHeight =
        = pDrvObj->createArgs.scParams.inHeight/2;

  pDrvObj->createArgs.scParams.inCropCfg.cropStartX = 0
  pDrvObj->createArgs.scParams.inCropCfg.cropStartY = 0

  pDrvObj->createArgs.scParams.inCropCfg.cropWidth
        = pDrvObj->createArgs.scParams.inWidth;

  pDrvObj->createArgs.scParams.inCropCfg.cropHeight
        = pDrvObj->createArgs.scParams.inHeight;

  pDrvObj->createArgs.scParams.inScanFormat = FVID2_SF_PROGRESSIVE;
  pDrvObj->createArgs.scParams.scConfig = NULL; // driver will use
default config
  pDrvObj->createArgs.scParams.scCoeffConfig = NULL; // driver will
load default co-effs

  pDrvObj->createArgs.vipParserInstConfig = NULL; // driver will use
 default VIP Parser config
  pDrvObj->createArgs.vipParserPortConfig = NULL; // driver will use
 default VIP Parser config
  pDrvObj->createArgs.cscConfig = NULL; // driver will use default
CSC settings

  pDrvObj->createArgs.inScanFormat = FVID2_SF_PROGRESSIVE;

  for(streamId=0; streamId <
pDrvObj->createArgs.numStream;streamId++)
  {
    for(chId=0; chId < pDrvObj->createArgs.numCh; chId++)
    {
      /* A utility API shown below is used to create unique
      channelNum for every channel in every stream, for a given
instance
      */
      pDrvObj->createArgs.channelNumMap[streamId][chId] =
        Vps_captMakeChannelNum(instId, streamId, chId);
```

```
    }
  }

  memset( & pDrvObj->cbPrm, 0, sizeof(pDrvObj->cbPrm));

  pDrvObj->cbPrm.cbFxn = set user callback function
  pDrvObj->cbPrm.appData = pDrvObj; // set user callback context

  pDrvObj->fvidHandle = FVID2_create(
                FVID2_VPS_CAPT_VIP_DRV,
                instId,
                & pDrvObj->createArgs,
                & pDrvObj->createStatus,
                & pDrvObj->cbPrm
              );

  if(pDrvObj->fvidHandle==NULL) {
  // error in driver instance create
    return -1;
  }

  return CaptureApp_allocAndQueueFrames(pDrvObj);
}


/*
  Allocate and queue frames to driver

  pDrvObj - capture driver information
*/
Int32 CaptureApp_allocAndQueueFrames(CaptureApp_DrvObj *pDrvObj)
{
  Int32 status;
  UInt16 streamId, chId, frameId, idx;
  FVID2_Format  format;
  FVID2_Frame   *frames;
  Vps_CaptRtParams  *rtParams;
  FVID2_FrameList   frameList;
  Vps_CaptOutInfo *pOutInfo;

  /* init frameList for list of frames that are queued per CH to driver
 */
  frameList.perListCfg = NULL;
  frameList.reserved = NULL;

  /* for every stream and channel in a capture handle */
  for(streamId=0; streamId < pDrvObj->createArgs.numStream;
```

```
streamId++)
  {
    for(chId=0; chId < pDrvObj->createArgs.numCh; chId++)
    {

      pOutInfo = & pDrvObj->createArgs.outStreamInfo[streamId];

      /* base index for pDrvObj->frames[] and pDrvObj->rtParams[]
 */
      idx =
VPS_CAPT_CH_PER_PORT_MAX*CAPTURE_APP_FRAMES_PER_CH*streamId
           + CAPTURE_APP_FRAMES_PER_CH*chId
              ;

      rtParams = & pDrvObj->rtParams[idx];
      frames = & pDrvObj->frames[idx];

      /* fill format with channel specific values  */
      format.channelNum = Vps_captMakeChannelNum(
                              pDrvObj->instId,
                              streamId,
                              chId
                           );
      format.width = input source width;
      format.height = input source height;
      format.pitch[0] = pOutInfo->pitch[0];
      format.pitch[1] = pOutInfo->pitch[1];
      format.pitch[2] = pOutInfo->pitch[2];
      format.fieldMerged[0] = FALSE;
      format.fieldMerged[1] = FALSE;
      format.fieldMerged[2] = FALSE;
      format.dataFormat = pOutInfo->dataFormat;
      format.scanFormat = FVID2_SF_PROGRESSIVE;
      format.bpp = FVID2_BPP_BITS8; /* ignored */


      /* alloc memory based on 'format'
        Allocated frame info is put in frames[]
        CAPTURE_APP_FRAMES_PER_CH is the number of buffers per channel
to
        allocate
      */
      VpsUtils_memFrameAlloc(format, frames,
CAPTURE_APP_FRAMES_PER_CH);

      /* Set rtParams for every frame in perFrameCfg  */
      for(frameId=0; frameId < CAPTURE_APP_FRAMES_PER_CH; frameId++)
```

```
      {
        frames[frameId].perFrameCfg = & rtParams[frameId];
        pFrames[frameId] = & frames[frameId];
      }

      /* Set number of frame in frame list  */
      frameList.numFrames = CAPTURE_APP_FRAMES_PER_CH;

      /* queue the frames in frameList
         All allocate frames are queued here as an example.
         In general atleast 2 frames per channel need to queued
         before starting capture,
         else frame will get dropped until frames are queued
      */
      status = FVID2_queue(pDrvObj->fvidHandle, & frameList,
streamId);
      assert(status==FVID2_SOK);
    }
  }

  return FVID2_SOK;
}
```

After FIVD2 create is called user should allocate the memory for frames to be captured and then queue the buffers to the driver using FVID2 queue. Atleast three buffers per channel need to be queued initially to the driver in order to receive frames without any frame drops. A utility function VpsUtils_memFrameAlloc() is provided to help user in their frame memory allocation. User is however free to use their own memory allocation function.

Internally the following happens when FVID2 create is called:

- Hardware resources like VIP parser ports, scalar, etc are allocated
- Software resources like semaphores, queues are allocated depending on the create parameters that are passed
- VIP hardware registers are initialized and VIP is made ready to begin capturing data,
- Data capture itself is not started during FVID2 create.

Run Phase

In this phase the driver can be used to start capture and continuously capture (dequeue) frame buffers from the driver and then process them and release (queue) them back to the driver.

Start and stop

Below API is used to start the capture. Once capture is started other FVID2 APIs can be used to dequeue and queue frame's continuously from the capture driver.

```
#include "ti/psp/vps/vps_capture.h"

status = FVID2_start(fvidHandle, NULL);
if(status!=FVID2_SOK) {
  // error in starting the capture
}
```

Below API is used to stop the capture. Capture can be started once again by using the FVID2 start API without having to create the driver once again.

```
#include "ti/psp/vps/vps_capture.h"

status = FVID2_stop(fvidHandle, NULL);
if(status!=FVID2_SOK) {
  // error in stopping the capture
}
```

Dequeueing-Queuing frames

Once the capture is started as described above, below API can be used to dequeue captured frames from the capture driver. Once capture is started it starts capturing data in the frame buffer's allocated and queued during create phase. Once a frame is captured completely, it queue's the captured frame to its "completed" frame queue. Now when user calls dequeue the captured frames are given to the user application.

A single dequeue call can be used to dequeue multiple captured frames from multiple channels associated with that handle. Similarly a single queue can used to return multiple frames from different channels associated with that handle back to driver.

Example: Non-blocking dequeue from stream 0 for a capture handle

The API used is a non-blocking API, i.e. API will return immediately with zero or more captured buffers.

```
#include "ti/psp/vps/vps_capture.h"

FVID2_FrameList       frameList;

status = FVID2_dequeue(fvidHandle, & frameList, 0, BIOS_NO_WAIT);
if(status!=FVID_SOK) {
  // error in dequeue-ing frames from capture handle
} else {
  // success, received 0 or more frames
  printf(" Received %d frames\n", frameList.numFrames);
}
```

Example: Dequeue from all active handles using a single API

The global VIP handle, fvidHandleVipAll, is created by user during system init and can be used to dequeue/queue frames from all active (created) capture handles.

```
#include "ti/psp/vps/vps_capture.h"

FVID2_FrameList       frameList;

status = FVID2_dequeue(fvidHandleVipAll, & frameList, 0,
BIOS_NO_WAIT);
if(status!=FVID_SOK) {
  // error in dequeue-ing frames from capture handle
} else {
  // success, received 0 or more frames
  printf(" Received %d frames\n", frameList.numFrames);
```

```
}
```

Example: Queue captured (dequeued) frames back to the driver

The frame dequeued would typically be processed by user application like encoding, scaling etc and once user is done with the frame, user application should queue the frames back to the driver as shown below. Instead of instance specifc handle shown below, the global VIP capture driver handle can also be used to queue the frame back to the correct driver instance without the user having to worry about which handle the frames belong to.

```
#include "ti/psp/vps/vps_capture.h"

FVID2_FrameList       frameList;

status = FVID2_queue(fvidHandle, & frameList, 0);
if(status!=FVID_SOK) {
  // error in queue-ing frames to capture handle
} else {
  // success
}
```

TIP

User should make sure to dequeue / queue frames from the capture handle at the required rate (frame-rate), else the capture driver may not have frames internally to write video data and it will then be forced to drop frames until a buffer is available.

Callback

A user callback can be registered during driver create which is then called by the driver whenever data is available at any of the channels, streams associated with the driver. User would typicall set a semaphore to wake up a task. The woken up task will then call dequeue API to get the newly captured frames. Dequeue should be called for every stream associated with the driver to get the captured frames, since the callback just indicates there is data but the data could be in any of the streams that are valid for the driver instance.

NOTE

The callback itself could be called from interrupt or SWI context, so user should use only APIs that are allowed in interrupt or SWI context inside the callback.

Understanding captured frame information

Once a frame is captured the FVID2 frame structure contains information about the captured frame. The captured information can retrieved as shown below. The example below assumes "channelNum" was created using the utility API Vps_captMakeChannelNum() during create.

```
#include "ti/psp/vps/vps_capture.h"

FVID2_FrameList       frameList;
FVID2_Frame          *pCurFrame;
Vps_CaptRtParams     *pCaptureRtParams;

Int32 frameId;

FVID2_dequeue(fvidHandleVipAll, & frameList, 0, BIOS_WAIT_FOREVER);
```

```
System_printf(" CAPTUREAPP: Received %d frame(s) \n",
frameList.numFrames);

for(frameId=0; frameId < frameList.numFrames; frameId++)
{
  pCurFrame = frameList.frames[frameId];

  pCaptureRtParams = (Vps_CaptRtParams*)pCurFrame->perFrameCfg;

  System_printf(" CAPTUREAPP: %d: time %d: ch %d:%d:%d: fid %d: %dx%d:
addr 0x%08x\n",
    frameId,
    pCurFrame->timeStamp,                       // timestamp in msecs
    Vps_captGetInstId(pCurFrame->channelNum),  // VIP instance ID
    Vps_captGetStreamId(pCurFrame->channelNum),// Stream ID
    Vps_captGetChId(pCurFrame->channelNum),    // channel ID
    pCurFrame->fid,                             // Even or Odd field
    pCaptureRtParams->captureOutWidth,      // captured frame width
    pCaptureRtParams->captureOutHeight,     // captured frame height
    pCurFrame->addr[0][0]                       // captured buffer
address for YUV422P format
               );
}

// process captured frames ...

...


FVID2_queue(fvidHandleVipAll, & frameList, 0);
```

TIP

Be careful to not modify the "FVID2_Frame.perFrameCfg" from the received (dequeued) frame when returning (queuing) the frame back to the driver. If FVID2_Frame.perFrameCfg has to be modified make sure user application sets it to a valid pointer in order to get captured frame width, height information from the driver, else set it to NULL.

Understanding FVID2_Frame.channelNum

Important

Be careful to not modify the "FVID2_Frame.channelNum" from the received (dequeued) frame when returning (queuing) the frame back to the driver. The FVID2 queue API uses "channelNum" to identify the VIP instance, stream and channel that the frame belongs to, in order to return it to the correct channel "free" queue.

The below description is valid only when during create, channel number was made using the utility API Vps_captMakeChannelNum(). In case user had created channel number using their own logic they need to apply a inverse logic in order to know the instance, stream, channel associated with the received frame .

The capture driver assigns a unique channel number to every video channel, stream that is being captured via any of the VIP ports. User application needs to be aware of this assignment when handling frames from different VIP ports .

"FVID2_Frame.channelNum" identifies the channel associated with a given frame. Given "FVID2_Frame.channelNum" user application can find out the VIP instance, stream and channel ID using the APIs shown in above example.

The table below shows the channel number assignment for different VIP ports:

VIP port channel number assignment

VIP Instance	Output Stream 0	Output Stream 1	Output Stream 2	Output Stream 3
VIP0 Port A	CH00 .. CH15	CH16 .. CH31	CH32 .. CH47	CH48 .. CH63
VIP0 Port B	CH64 .. CH79	CH80 .. CH95	CH96 .. CH111	CH112 .. CH127
VIP1 Port A	CH128 .. CH143	CH144 .. CH159	CH160 .. CH175	CH176 .. CH191
VIP1 Port B	CH192 .. CH207	CH208 .. CH223	CH224 .. CH239	CH240 .. CH255

Control IOCTLs supported

Frame skip control IOCTL_VPS_CAPT_SET_FRAME_SKIP

User can program a frame skip mask per channel to selectively skip frames. In this way user can control the frame-rate at which they want the data to be captured. When a frame is skipped, it is not written to DDR so that will also result in DDR bandwidth reduction.

This IOCTL can be called even while capture is running and frame-skip mask can be changed dynamically while capture is running.

An example is given below:

```
#include "ti/psp/vps/vps_capture.h"

Vps_CaptFrameSkip frameSkip;

// channelNum is the one that was specified by user during create in
channelNumMap[][]
frameSkip.channelNum = createArgs.channelNumMap[streamId][chId];

// Example: for full frame-rate
frameSkip.frameSkipMask = 0;

// Example: for 1/2 frame-rate
frameSkip.frameSkipMask = 0x2AAAAAAA;

status = FVID2_control(
            fvidHandle,
            IOCTL_VPS_CAPT_SET_FRAME_SKIP,
            & frameSkip,
            NULL
          );
```

Digital Pan, Crop, Down-scale control IOCTL_VPS_CAPT_SET_SC_PARAMS

This IOCTL is valid for streams where Vps_CaptCreateParams.outInfo[x].scEnable = TRUE during create.

This IOCTL allows user to select the scalar input area and output area and thus effectively doing digital crop and/or down-scale and pan.

User must make sure output width and height do not exceed actual buffer size allocated by user.

- **Scaler coefficient load:** If enableCoeffLoad is specified as FALSE in the `Vps_CaptScParams` provided, this API shall not load scaler co-efficients. If enableCoeffLoad is specified as TRUE, scaler coefficients shall also be loaded during this IOCTL.
  - If the user provides specific scaler coefficients in `scParams.scCoeffConfig`, these are used for programming the scaler. If not provided, the driver internally calculates the scaling factor for the provided scaler params.
  - The driver checks if the scaling factor has changed with respect to the current scaling factor.
  - If the scaling factor, and hence the horizontal & vertical coefficients (for polyphase filter) are the same, nothing extra needs to be done.
  - If the scaling factor, and hence the horizontal or vertical coefficients (for polyphase filter) have changed, the new scaler coefficients are fetched.
    - To enable loading the new scaler coefficients, the VIP instance is then stopped, the scaler coefficients programmed, the VIP instance is reset, and then restarted. This results in a maximum of two frame drops.
  - Even if the scaler is initially in bypass at create time, scaler coefficients are loaded with default values internally.
  - If scaler is brought out of bypass in this ioctl, if `enableCoeffLoad` is specified as FALSE, the default coefficients will continue to be used. If `enableCoeffLoad` is specified as TRUE, new suitable coefficients shall get loaded at that time.
  - If scaler is placed in bypass through this ioctl, scaler coefficient load is not done even if `enableCoeffLoad` is TRUE.

If scaler coefficients are not to be loaded, this IOCTL effect will take place from next frame that is captured.

ImportantUpscaling is NOT supported during in-line scaling during capture and should NOT be done. This API can be called anytime after FVID2 create. It can be called even when capture is running

An example is shown below:

```
#include "ti/psp/vps/vps_capture.h"

Vps_CaptScParams scParams;

scParams.inWidth = createArgs.scParams.inWidth;
scParams.inHeight = createArgs.scParams.inHeight;

scParams.inCropCfg.cropStartX  = 0;
scParams.inCropCfg.cropStartY  = 0;
scParams.inCropCfg.cropWidth   = createArgs.scParams.inWidth;
scParams.inCropCfg.cropHeight  = createArgs.scParams.inHeight;

/* setting output wxh to original w/4 x h/4
   this is just a example
*/
scParams.outWidth  = createArgs.scParams.inWidth/4;
```

```
scParams.outHeight = createArgs.scParams.inHeight/4;

scParams.inScanFormat = FVID2_SF_PROGRESSIVE;
scParams.scConfig = NULL; /* when NULL driver uses default SC config */
scParams.scCoeffConfig = NULL; /* IGNORED BY IOCTL */
scParams.enableCoeffLoad = TRUE; /* Enable scaler coefficient load */

FVID2_control(
    fvidHandle,
    IOCTL_VPS_CAPT_SET_SC_PARAMS,
    & scParams,
    NULL
  );
```

Get Channel Status IOCTL_VPS_CAPT_GET_CH_STATUS

This IOCTL allows user to get channel related information as detected by the hardware, like channel data width, height, video detect.

Width and height that is returned is the width and height of the last captured frame that hardware has detected.

Video detect status is calculated based on last received frame timestamp and expected frame interval. If a frame is not received in the given frame interval, then its considered as video is not detected.

Typically usage of this would be to periodically call this API from user context, say every 10ms or 30ms. User could then use this API to know detected video width and height and then allocate and queue buffers to the driver.

An example is shown below:

```
#include "ti/psp/vps/vps_capture.h"

/*
  Check video detect status using IOCTL
*/
int status, chId, streamId;
Vps_CaptChGetStatusArgs chStatusArgs;
Vps_CaptChStatus chStatus

/* for all streams and channels */
for(streamId=0; streamId < createArgs.numStream; streamId++)
{
  for(chId=0; chId < createArgs.numCh; chId++)
  {

    chStatusArgs.channelNum = createArgs.channelNumMap[streamId][chId];

    /* expected frame capture interval between two frames/field in
msecs  */
    chStatusArgs.frameInterval = 16;

    /* get video detect status  */
    status = FVID2_control(
```

```
                fvidHandle,
                IOCTL_VPS_CAPT_GET_CH_STATUS,
                & chStatusArgs,
                & chStatus
              );

    if(chStatus.isVideoDetected)
    {
      /* video detect, print video info */
      System_printf(" DETECT = %d: %dx%d\n",
                  chStatus.isVideoDetected,
                  chStatus.captureOutWidth,
                  chStatus.captureOutHeight
               );
    }
  }
}
```

Set Buffer Storage format IOCTL_VPS_CAPT_SET_STORAGE_FMT

This ioctl is used to set the buffer storage format for the interlaced capture. Application can configure driver to capture individual fields or both fields (frame) using this ioctl. For the frame capture in the interlaced input format, both fields can be merged or it can be separate. In frame capture mode application needs to queue buffer for both the fields using `FVID2_queue` and application gets call back once both fields are captured. In field capture mode application needs to queue buffer

An example is given below:

```
#include "ti/psp/vps/vps_capture.h"

Vps_CaptStoragePrms storagePrms;

/* chNum is ignored configuration applies for all channels/streams of
 * of the handle
 */
if (TRUE == gCaptureApp_ctrl.utParams.fieldMerged)
{
    storagePrms.chNum = 0;
    storagePrms.bufferFmt = FVID2_BUF_FMT_FRAME; /* FVID2_BUF_FMT_FIELD
 for field capture */
    storagePrms.fieldMerged = TRUE;
    status = FVID2_control (pDrvObj->fvidHandle,
                            IOCTL_VPS_CAPT_SET_STORAGE_FMT,
                            &storagePrms, NULL  );
    GT_assert( GT_DEFAULT_MASK, status == FVID2_SOK );
}
```

Other IOCTLs

Followingare some other IOCTLs that can be used by user for different purposes as shown in below table

IOCTL	Applicable Driver Handle	Purpose
IOCTL_VPS_CAPT_RESET_VIP0	Global Handle ONLY	This is used to reset whole VIP0 block. This MUST be called before calling FVID2 create for the VIP instance. Refer to API Guide for more details.
IOCTL_VPS_CAPT_RESET_VIP1	Global Handle ONLY	Same as IOCTL_VPS_CAPT_RESET_VIP0 excpet that it applies to VIP1 instance
IOCTL_VPS_CAPT_PRINT_ADV_STATISTICS	Global Handle ONLY	Prints advanced debug and statistical information like frames captured, fps etc. Used for Advanced debug ONLY
IOCTL_VPS_CAPT_CHECK_OVERFLOW	Global Handle ONLY	Checks if any VIP instance has overflowed. Refer to API Guide for more details.
IOCTL_VPS_CAPT_RESET_AND_RESTART	Global Handle ONLY	In any any VIP port is overflowed, this IOCTL can be used to reset and restart capture associated with that VIP instance. Refer to API Guide for more details.
IOCTL_VPS_CAPT_GET_STORAGE_FMT	Instance specific Handle	This ioctl let application know, driver is set for Field based capture of Frame based capture for interlaced inputs.

Delete Phase

In this phase FVID2 delete API is called to free all resources allocated during capture. Make sure capture is stopped using FVID2_stop() before deleting a capture instance. Once a capture handle is deleted the resources free'ed by that capture handle could be used when another capture driver or other related driver is opened.

The FVID2 delete API call is shown below:

```
#include "ti/psp/vps/vps_capture.h"

FVID2_delete(fvidHandle, NULL);
```

System De-init Phase

In this phase VIP capture sub-system is de-initialized. Here all resources acquired during system initialization are free'ed. Make sure all capture handles are deleted before calling this API. VIP sub-system de-init happens as part of overall FVID2 system de-init. Typically this is done during system shutdown.

The global VIP capture handle, if opened earlier, should also be deleted before called FVID2 de-init

```
#include "ti/psp/vps/vps_capture.h"

Void mySysDeInit() {

  /* Delete global VIP capture handle */
  FVID2_delete(fvidHandleVipAll, NULL);

  /* FVID2 system de-init */
  FVID2_deInit(NULL);
}
```

Sample application

This section shows how to configure and run the sample application for VIP capture. The sample application source code is located at the below path.

`pspdrivers_\packages\ti\psp\examples\common\vps\capture\captureVip`

Running the sample application

To run the sample application, load and run the `$(hdvpss_install_folder)\build\bin\$platform\m3vpss\whole_program_debug\hdvpss_examples_c` onto HDVPSS-M3 processor of the TI EVM for TI816x or TI814x via CCS v4 and above
This application will detect the platform board/EVM and then execute different capture modes depending on the detect platform board/EVM.

Currently following platform specific boards/EVMs and capture combinations are supported by the capture example.

	TI816x	**TI814x**
Video Security (VS) EVM	4x TVP5158 MULTI-CH capture	2x TVP5158 MULTI-CH capture
Video Conferencing (VC) EVM	2x SII9135 + 1x TVP7002 (muxed with 1x SII9135) SINGLE-CH capture	1x SII9135

Sample output printed on the CCS console for TI816x VS EVM is shown below:

```
 === HDVPSS Clocks are enabled ===
 === HDVPSS is fully functional ===
 === HDVPSS module is not in standby ===
 === I2C1 Clk is active ===
  CAPTUREAPP: HDVPSS Drivers Version String: HDVPSS_01_00_01_25
 *** VPDMA Firmware Loading... ***
 VPDMA Firmware Address = 0x921a2fc0
 VPDMA Load Address     = 0x4810d004
 VPDMA Firmware Version = 0x4d000185
 VPDMA List Busy Status = 0x00000000
 *** VPDMA Firmware Load Success ***

  I2C1: Passed for address 0x20 !!!
  I2C1: Passed for address 0x21 !!!
  I2C1: Passed for address 0x39 !!!
  I2C1: Passed for address 0x3d !!!
  I2C1: Passed for address 0x58 !!!
  I2C1: Passed for address 0x5a !!!
  I2C1: Passed for address 0x5c !!!
  I2C1: Passed for address 0x5e !!!
  I2C1: Passed for address 0x60 !!!
  CAPTUREAPP : Detected [4x TVP5158 VS] Board !!!
  CAPTUREAPP: CaptureApp_init() - DONE !!!
  CAPTUREAPP: Loop 1 of 1 !!!
  CAPTUREAPP: HANDLES 4: MODE 0002 : CH 4: RUN COUNT 10:
OUTPUT:1:12292 !!!
  CAPTUREAPP: 0: CaptureApp_create() - DONE !!!
```

```
CAPTUREAPP: VIP 0: VID DEC 0400 (0x58): 5158:0002:0000
CAPTUREAPP: Detect video in progress for inst 0 !!!
TVP5158: 0x58: Downloading patch ...
TVP5158: 0x58: Downloading patch ... DONE !!!
TVP5158: 0x58: 5158:0002:0124
CAPTUREAPP: Detected video at CH0 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH1 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH2 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH3 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detect video Done !!!
CAPTUREAPP: 1: CaptureApp_create() - DONE !!!
CAPTUREAPP: VIP 2: VID DEC 0400 (0x5a): 5158:0002:0000
CAPTUREAPP: Detect video in progress for inst 2 !!!
TVP5158: 0x5a: Downloading patch ...
TVP5158: 0x5a: Downloading patch ... DONE !!!
TVP5158: 0x5a: 5158:0002:0124
CAPTUREAPP: Detected video at CH0 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH1 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH2 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH3 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detect video Done !!!
CAPTUREAPP: 2: CaptureApp_create() - DONE !!!
CAPTUREAPP: VIP 1: VID DEC 0400 (0x5c): 5158:0002:0000
CAPTUREAPP: Detect video in progress for inst 1 !!!
TVP5158: 0x5c: Downloading patch ...
TVP5158: 0x5c: Downloading patch ... DONE !!!
TVP5158: 0x5c: 5158:0002:0124
CAPTUREAPP: Detected video at CH0 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH1 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH2 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH3 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detect video Done !!!
CAPTUREAPP: 3: CaptureApp_create() - DONE !!!
CAPTUREAPP: VIP 3: VID DEC 0400 (0x5e): 5158:0002:0000
CAPTUREAPP: Detect video in progress for inst 3 !!!
TVP5158: 0x5e: Downloading patch ...
TVP5158: 0x5e: Downloading patch ... DONE !!!
TVP5158: 0x5e: 5158:0002:0124
CAPTUREAPP: Detected video at CH0 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH1 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH2 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detected video at CH3 (720x240@59Hz, 1)!!!
CAPTUREAPP: Detect video Done !!!
CAPTUREAPP: Starting capture ... !!!
CAPTUREAPP: Starting capture ... DONE !!!
CAPTUREAPP: Capture in progress ... DO NOT HALT !!!
CAPTUREAPP: Stopping capture ... !!!
```

```
  CAPTUREAPP: Stopping capture ... DONE !!!

  Execution Statistics
  ====================

  Execution time    : 11.65 s
  Total field Count : 10684 (965 fields/sec)
  Avg CPU Load      : 5 %

  *** Capture Driver Advanced Statistics ***

  VIP Parser Reset Count : 0

      |   Total    Even     Odd  Total  Even   Odd  Min /  Max  Min /
Max Dropped FrmErr
  CH  |  Fields  Fields  Fields    FPS   FPS   FPS       Width
Height  Fields (DescErr)

-------------------------------------------------------------------------------
  000 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  001 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  002 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  003 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  100 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  101 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  102 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  103 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  200 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  201 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  202 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  203 |     668     334     334     60    30    30  720 /  720  243 /
244       0      0(0)
  300 |     667     333     334     60    30    30  720 /  720  243 /
244       0      0(0)
  301 |     667     333     334     60    30    30  720 /  720  243 /
244       0      0(0)
```

```
  302 |     667     333     334     60    30    30  720 /  720  243 /
244       0      0(0)
  303 |     667     333     334     60    30    30  720 /  720  243 /
244       0      0(0)

  *** Capture List Manager Advanced Statistics ***

  List Post Count        : 5516
  List Stall Count       : 0
  List Post Time (ms)    : Max = 0, Min = 0, Avg = 0, Total = 0
  Error descriptor count : 0
  Recv  descriptor count : 0
  Extra descriptor programmed count : 0


  VIP and VPDMA registers,
  VIP0 : FIQ_STATUS  : 0x4810551c = 0x00001400
  VIP1 : FIQ_STATUS  : 0x48105a1c = 0x00001400
  VPDMA: LIST_BUSY   : 0x4810d00c = 0x00000000
  VPDMA: PERF_MON32 = 0xb3350000, PERF_MON33 = 0xa332496d, PERF_MON34 =
 0x81000001, PERF_MON35 = 0x13150000


  CAPTUREAPP: 0: CaptureApp_delete() - DONE !!!
  CAPTUREAPP: 1: CaptureApp_delete() - DONE !!!
  CAPTUREAPP: 2: CaptureApp_delete() - DONE !!!
  CAPTUREAPP: 3: CaptureApp_delete() - DONE !!!
  CAPTUREAPP: ERROR COUNT 0: HANDLES 4: MODE 0002 : CH 4: RUN COUNT
10: !!! - DONE

  24870: LOAD: CPU: 5% HWI: 0%, SWI:0%

  CAPTUREAPP: CaptureApp_deInit() - DONE !!!
```

TI81xx-external video drivers

Introduction

External Video drivers control the video devices, which are external to the HDVPSS. These devices include TVP5158, TVP7002, Sil9022, SiL9135 etc. These video drivers are exposed to the application by FVID2 interface.

Sil9022A HDMI Transmitter

Sil9022A supports High Definition Multimedia Interface. It support YUV422, YUV444 and RGB as the input and output pixel format. It supports embedded as well as seprate sync format. It has internal DE generator to support non-embedded sync format. Its output is compatible with HDMI, HDCP and DVI. It supports resolution upto 1080p and it it pre-programmed with HDCP keys and has completely self-sequencing HDCP detection and authentication, including SHA-1 for repeaters.

Sil9022A is present on both VS as well as VC daughter cards. It is exposed to application throug FVID2 interface. Sil9022A FVID2 driver can be opened using FVID2_VPS_VID_ENC_SII9022A_DRV driver Id and 0 as the instance Id.

Important

The interfaces defined in this file is bound to change. Kindly treat the interfaces as work in progress. Release notes/user guide list the additional limitation/restriction of this module/interfaces.

Features Supported

- Supports FVID2 interface
- Supports 720p@60, 1080I@60, 1080p@30 and 1080p@60 modes
- Supports ioctls to start and stop hdmi output, to get the status of the hot plug detection event, to get the chip Id.

Features Not Supported

- Selection on input and output format
- Selection of input format
- DVI as output
- HDCP

Limitations/Issues

- None

Software Application Intefaces

Since this driver is used to control HDMI9022A only, it is not a streaming driver. The driver operation can be partitioned into the below phases:

- System Init Phase: Here the driver sub-system is initialized
- Create Phase: Here the driver handle is created or instantiated
- Run Phase: Here the drive is started or stopped
- Delete Phase: Here the driver handle or instance is deallocated
- System De-init Phase: Here the driver sub-system is de-initialized

The subsequent sections describe each phase in detail.

System Init Phase

The HDMI9022A sub-system initialization happens as part of overall HDVPSS system init and platform init. This API must be the the first API call before making any other FVID2 calls. Below section lists all the APIs which are part of the System Init phase.

FVID2 Init

```
Int32 FVID2_init(Ptr args);
```

args - `NULL` currently not used.

Platform Init

```
Int32 VpsUtils_platformEvmInit();
```

System Create Phase

In this phase user application opens or creates a driver instance. Any number of instances can be created for this hdmi driver. After opening the driver, status can be obtained and mode can be set in the driver.

FVID2 Create

This API is used to open the HDMI 9022A driver. This is a blocking call and it returns the handle which is to be used in subsequent call to this driver.

```
FVID2_Handle FVID2_create(UInt32 drvId,
                          UInt32 instanceId,
                          Ptr createArgs,
                          Ptr createStatusArgs,
                          const FVID2_CbParams *cbParams);
```

drvId - `FVID2_VPS_VID_ENC_SII9022A_DRV` HDMI 9022A Driver ID. Use this ID to open HDMI9022A driver. Details can be found in UserGuide

instanceId - `0` Instance 0 of the HDMI 9022A driver.

createArgs - Pointer to `Vps_VideoEncoderCreateParams` structure containing valid create params. This parameter must not be null.

createStatusArgs - Pointer to `UInt32` return value where the driver returns the actual return code for create function. This parameter should not be NULL.

cbParams - Since there is no callback from the display controller, this parameters should be set to NULL.

FVID2 Control - Get Detailed Chip ID

This is used to issue a control command to the driver. IOCTL_VPS_SII9022A_GET_DETAILED_CHIP_ID ioctl is used to get the detailed chip id.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SII9022A_GET_DETAILED_CHIP_ID` ioctl.

cmdArgs - Pointer to `Vps_HdmiChipId` structure containing revision id for device, hdcp, TPI.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control Get hot plug Status

This is used to issue a control command to the driver. IOCTL_VPS_SII9022A_QUERY_HPD ioctl is use to query status of the hot plug detect event.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_SII9022A_QUERY_HPD` ioctl.

cmdArgs - Pointer to `Vps_SiI9022aHpdPrms` structure containing event for hpd, bus error.

cmdStatusArgs - Not used currently. This parameter should be set to NULL.

FVID2 Control - Set Mode

This is used to issue a control command to the driver. IOCTL_VPS_VIDEO_ENCODER_SET_MODE ioctl is used to set mode in the HDMI9022A.

```
Int32 FVID2_control(FVID2_Handle handle,
                    UInt32 cmd,
                    Ptr cmdArgs,
                    Ptr cmdStatusArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmd - `IOCTL_VPS_VIDEO_ENCODER_SET_MODE` ioctl.

cmdArgs - Pointer to `Vps_SiI9022aModeParams` structure containing mode parameters. **cmdStatusArgs** - Not used currently. This parameter should be set to NULL.

System Run Phase

This phase is used to start or stop the output from HDMI transmitter

FVID2 Start

This API is used by the application to start the output from HDMI9022A. This is a blocking call and returns after starting output. This cannot be called from ISR context.

```
Int32 FVID2_start(FVID2_Handle handle, Ptr cmdArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmdArgs - Not used currently. This parameter should be set to NULL.

FVID2 Stop

This API is used by the application to stop the output of HDMI9022A. This is a blocking call and returns after stopping the HDMI9022A output. This cannot be called from ISR context.

```
Int32 FVID2_stop(FVID2_Handle handle, Ptr cmdArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

cmdArgs - Not used currently. This parameter should be set to NULL.

Delete Phase

In this phase FVID2 delete API is called to close the driver instance.

FVID2 Delete

This API is used to close the HDMI9022A driver. This is a blocking call and returns after closing the handle.

```
Int32 FVID2_delete(FVID2_Handle handle, Ptr deleteArgs);
```

handle - Driver handle returned by create function call. This parameter should not be NULL.

deleteArgs - Not used currently. This parameter should be set to NULL.

System De-init Phase

In this phase, HDMI9022A driver is de-initialized. Here all resources acquired during system initialization are free'ed. Make sure all driver instances deleted before calling this API. HDMI9022A de-init happens as part of overall FVID2 system and platform de-init. Typically this is done during system shutdown.

Platform De-init

```
Int32 VpsUtils_platformEvmDeInit();
```

args - Not used

FVID2 De-init

```
Int32 FVID2_deInit(Ptr args);
```

args - Not used

UserGuideHdvpssIntegExample

Integration Examples

Link-chain Framework

This framework is developed (as an example) so that it becomes easier for the individual modules to integrate themselves with others, create a chain of different modules and test them together for a particular use case. It is based on few assumptions and may not be suitable for all possible use cases.

A link is a stand-alone independent task which is created on top of a specific module (like capture, scalar, noise filter etc) and is meant to perform a predefined operation. For e.g. the display link will only take care of the display related functionalities, a noise-filter link will only look towards noise filtering etc.

A chain is a set of links joined together to perform a complete sequence of operations desired for the specific use case. For e.g. capture and display links can be joined to form a chain which will capture frames from the input source and display them on the connected output.

During initialization, each link registers itself with this framework and then creates its own individual task. This task listens to the incoming messages and takes the appropriate actions.

Each link accepts messages from the previous link(s) (the only exception is capture link) and sends output to the connected link(s) (an exception is display). The link is informed by its predecessor link once the data is available for its use. The link can then start processing the data and once it is done, it informs the next link. The next connected link in the chain will then start processing the data and the process continues.

Link-Chains Examples

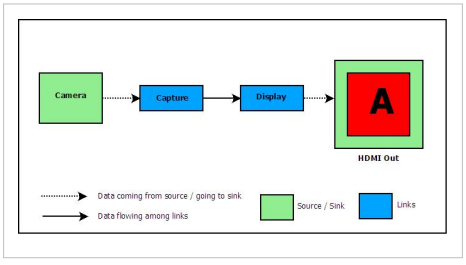
Below are a few examples of chains having different links with detailed diagrams and performing various tasks.

The chains and links implementation can be found in **\packages\ti\psp\examples\common\vps\chains** folder.

Main source file for all chains is: **src\chains_main.c**

NOTE: The examples shown in this section is not a exhaustive list of link-chains that are actually implemented in the HDVPSS chains examples. See next section for list of all links-chains that are implemented

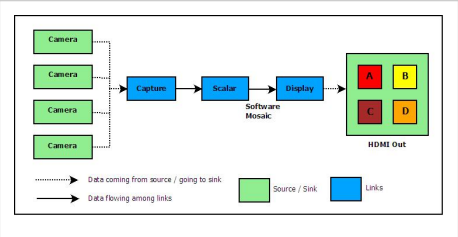
Single-channel Capture + Display



This is the simplest chain which uses capture and display links. Capture link takes single input from the source (camera) and informs the display link once the frames are captured. Display link then configures the attached display (for e.g. HDMI) and starts displaying captured frames from a single channel on the same. Both capture and display links operate in single channel mode here.

Source file: ***src\chains_singleChCaptureSii9135.c***

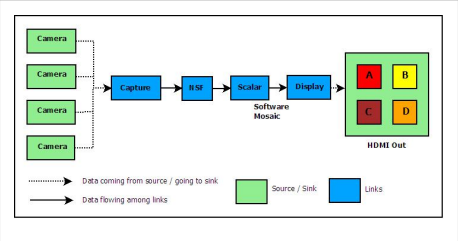
Multi-channel Capture + Scalar + Display



This chain uses capture, scalar and display links. Capture link takes multiple inputs from various sources and passes the captured frames to the scalar link. Scalar link then upscales/downscales the incoming frames as per the configuration parameters and performs software mosaic on them, resulting in a single frame having all the selected input frames. Scalar link then sends this output frame to the display link which, after configuring the attached display, starts displaying these frames on the same.

Source file: ***src\chains_multiChCaptureNsf.c***

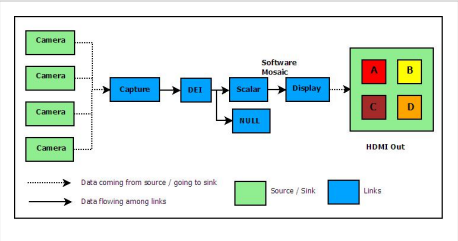
Multi-channel Capture + Noise-filter + Scalar + Display



This chain uses capture, noise-filter, scalar and display links. Capture link takes multiple inputs from various sources and passes the captured frames to the noise filter link. This link does the noise-filtering on incoming frames as per its configuration and sends them to the next link – scalar – in the chain. Scalar link then upscales/downscales the incoming frames as per the configuration parameters and performs software mosaic on them, resulting in a single frame having all the selected input frames. Scalar link then sends this output frame to the display link which, after configuring the attached display, starts displaying these frames on the same.

Source file: ***src\chains_multiChCaptureNsf.c***

Multi-channel Capture + De-interlacer + Scalar + Display

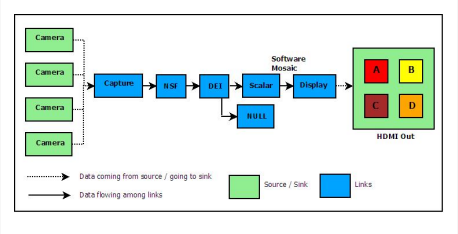


This chain uses capture, de-interlacer, scalar and display links. Capture link takes multiple inputs from various sources and passes the captured frames to the de-interlacer link. This link converts the interlaced video content from multiple channels to progressive form. This link has two outputs: one before the scalar in the DEI block and other after the scalar in the DEI block. Here, the first non-scaled output is sent to the scalar link and the other output is sent to the NULL link which acts as a dummy link and returns those frames immediately to the DEI link for further usage. Scalar link then upscales/downscales the incoming frames as per the configuration parameters and performs software mosaic on them, resulting in a single frame having all the selected input frames. Scalar link then sends this

output frame to the display link which, after configuring the attached display, starts displaying these frames on the same.

Source file: ***src\chains_multiChCaptureNsfDei.c***

Multi-channel Capture + Noise-Filter (NSF) + De-interlacer (DEI) + Scalar + Display



This chain exercises most of the available links - capture, noise-filter, de-interlacer, scalar and display links. Capture link takes multiple inputs from various sources and passes the captured frames to the noise-filter link. This link does the noise-filtering on incoming frames as per its configuration and sends them to the next link – de-interlacer – in the chain. This link converts the interlaced video content from multiple channels to progressive form. This link has two outputs: one before the scalar in the DEI block and other after the scalar in the DEI block. Here, the first non-scaled output is sent to the scalar link and the other output is sent to the NULL link which acts as a dummy link and returns those frames immediately to the DEI link for further usage. Scalar link then upscales/downscales the incoming frames as per the configuration parameters and performs software mosaic on them, resulting in a single frame having all the selected input frames. Scalar link then sends this output frame to the display link which, after configuring the attached display, starts displaying these frames on the same.

Source file: ***src\chains_multiChCaptureNsfDei.c***

Link - Chain Actual Samples in HDVPSS driver package

The following link-chains are implemented in HDVPSS driver package

Platforms Supported	Applicable Board	Menu Option in Chains Example	Main Source File	Data Flow	Additional Comments
TI816x TI814x TI8107	VS	1: Single CH Capture + Scale + Display (2CH 2x TVP5158, NTSC, YUV420SP)	chains_tvp5158NonMuxCapture.c	- Non-mux D1 capture via 2x TVP5158 in YUV420SP dual output format - Output from capture scaled from YUV420SP to YUV422I using SC5 and output arranged in 2x2 layout - Displayed on a 1080p60 Display (On-Chip or Off-HDMI)	

TI816x TI814x TI8107	VS	2: Multi CH Capture + Scale + Display (TVP5158, NTSC, YUV422I)	chains_multiChCaptureNsf.c	- 4CH D1 muxed capture via 4x TVP5158 in YUV422I output format - Output from capture scaled from YUV422I to YUV422I using SC5 and output arranged in 4x4 layout - Displayed on a 1080p60 Display (On-Chip or Off-HDMI)	
TI816x TI814x TI8107	VS	3: Multi CH Capture + NSF + Scale + Display (TVP5158, NTSC, YUV422I)	chains_multiChCaptureNsf.c	- 4CH D1 muxed capture via 4x TVP5158 in YUV422I output format - Output from capture noise filtered (NSF) (YUV422I to YUV420SP conversion while doing NSF) - Output from NSF scaled from YUV420SP to YUV422I using SC5 and output arranged in 4x4 layout - Displayed on a 1080p60 Display (On-Chip or Off-HDMI)	NSF mode full enable or bypass/disable can be selected via settings menu

TI816x TI814x TI8107	VS	4: Multi CH Capture + DEI + Scale + Display (4CH 1x TVP5158, NTSC, YUV422I)	chains_multiChCaptureNsfDei.c	- 4CH D1 muxed capture via 1x TVP5158 in YUV422I output format - Output from capture deinterlaced (DEI) and 1:1 scaled using DEI-SC as well as 1:1 scaled via VIP-SC - Output from DEI-SC scaled from YUV422I to YUV422I using SC5 and output arranged in 2x2 layout - Displayed on a 1080p60 Display (On-Chip or Off-HDMI)	DEIH or DEI can be selected via settings menu The DEI output to be scaled and displayed can be DEI-SC output or VIP-SC output depending on option selected via settings menu
TI816x TI814x TI8107	VS	5: Multi CH Capture + NSF + DEI + Scale + Display (8CH 2x TVP5158, NTSC, YUV422I)	chains_multiChCaptureNsfDei.c	- 4CH D1 muxed capture via 2x TVP5158 in YUV422I output format - Output from capture noise filtered (NSF) (YUV422I to YUV420SP conversion while doing NSF) - Output from NSF deinterlaced (DEI) and 1:1 scaled using DEI-SC as well as 1:1 scaled via VIP-SC - Output from DEI-SC scaled from YUV422I to YUV422I using SC5 and output arranged in 2x2 layout - Displayed on a 1080p60 Display (On-Chip or Off-HDMI)	NSF mode full enable or bypass/disable can be selected via settings menu DEIH or DEI can be selected via settings menu The DEI output to be scaled and displayed can be DEI-SC output or VIP-SC output depending on option selected via settings menu

TI816x	VS	6: Multi CH Capture + DEI + Mosaic Display (8CH 2x TVP5158, NTSC, YUV422I)	chains_multiChSystemUseCase.c	- 4CH D1 muxedcapture via 2x TVP5158 in YUV422I output format - Output from capture deinterlaced (DEI) and scaled using DEI-SC for 8CH layout - Output from DEI shown in 8CH mode via 1080p60 mosaic display	DEIH or DEI can be selected via settings menu
TI816x	VS	7: Multi CH System Use Case (16CH 4x TVP5158, NTSC, YUV422I)	chains_multiChSystemUseCase.c	- 4CH D1 muxed capture via 4x TVP5158 in YUV422I output format - Output from capture noise filtered (NSF) (YUV422I to YUV420SP conversion while doing NSF) - Output from NSF deinterlaced (DEIH+DEI) and scaled using DEI-SC for 4x4 layout - Output from DEIH+DEI shown in 16CH mode via 1080p60 mosaic display	NSF mode full enable or bypass/disable can be selected via settings menu

TI816x	VS	8: Multi CH System Use Case - II (16CH 4x TVP5158, NTSC, YUV422I)	chains_multiChSystemUseCase2.c	- 4CH D1 muxed capture via 4x TVP5158 in YUV422I output format - Output from capture noise filtered (NSF) (YUV422I to YUV420SP conversion while doing NSF) - Output from NSF deinterlaced (DEIH+DEI) and 1:1 scaled using DEI-SC as well as 1:1 scaled via VIP-SC - Output from DEIH-VIP-SC + DEI-VIP-SC scaled from YUV420SP to YUV422I using dual instances of SC5 and output arranged in 8CH layout - Displayed on a dual 1080p60 Display (On-Chip + Off-HDMI)	This option needs additional memory so will not work with default memory map in HDVPSS examples. Need to modify memory map in config.bld file and give more memory in vpsutils_mem.h (needs 350MB) to frame buffers in order to make this work
TI816x TI814x	VS	9: Single CH Capture + DEI + Display (Full screen DEI) (1CH 1x TVP5158, NTSC, YUV422I)	chains_singleChCaptureNsfDeiTvp5158.c	- Non-mux D1 capture via 1x TVP5158 in YUV420SP dual output format - Output from capture noise filtered (NSF) (YUV422I to YUV420SP conversion while doing NSF) - Output from NSF deinterlaced (DEI) and scaled using DEI-SC for 1080p output - Displayed on a 1080p60 Display (On-Chip or Off-HDMI)	NSF mode full enable or bypass/disable can be selected via settings menu DEIH or DEI can be selected via settings menu

TI816x TI814x	VC	1: Single CH Capture + Display (1x SII9135 16b, 1080P60, YUV422I)	chains_singleChCaptureSii9135.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 1080p60 source) - Capture Output 1: Unscaled YUV422I output - Capture Output 2: Scaled 800x450 YUV422I output - Display, 1080p60 - Shows Capture Output 1 for 11secs and Capture Output 2 for 11 secs and so on	

TI816x TI814x	VC	2: Single CH Capture + SC + Display (1x SII9135 16b, 1080P60, YUV420SP)	chains_singleChCaptureSii9135.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 1080p60 source) - Capture Output 1: Unscaled YUV420SP output - Capture Output 2: Scaled 800x450 YUV420SP output - SC - (SC5) - Takes Capture Output 1 as input and outputs 1:1 SC YUV422I output - Does this for 11secs and then flips to Capture Output 2 as input and so on - Display, 1080p60 - Shows YUV422I output from SC	
TI816x TI814x	VC	3: Single CH Capture + NSF + DEI + Display (1x TVP7002 16b, 1080i60, YUV422I)	chains_singleChCaptureNsfDeiTvp7002.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture outputs YUV422I from VIP0 PortA (via TVP7002, expects 1080i60 input) - NSF takes capture output and converts it to YUV420 - DEI takes NSF output and converts it to YUV422 - Display takes DEI output and displays it	

TI816x TI814x	VC	4: Single CH Capture + DEI + Display (1x TVP7002 16b, 1080i60, YUV420SP)	chains_singleChCaptureNsfDeiTvp7002.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture outputs YUV420SP from VIP0 PortA (via TVP7002, expects 1080i60 input) - DEI takes Capture output and converts it to YUV422 - Display takes DEI output and displays it	
TI816x TI814x	VC	5: Single CH Capture + Display (1x TVP7002 24b, 1080i60, YUV422I)	chains_singleChCaptureTvp7002DisSync.c	- Capture operates in 24-bit RGB discrete sync mode - Capture from VIP0 PortA (via TVP7002, expects 1080i60 source) - CSC converts RGB to YUV - Capture Output 1: Unscaled YUV422I output - Capture Output 2: Scaled 960x540 YUV422I output - Display, 1080p60 - Shows Capture Output 1 for 11secs and Capture Output 2 for 11 secs and so on	

TI816x TI814x	VC	6: Single CH Capture + SC + Display (1x TVP7002 24b, 1080i60, YUV422SP)	chains_singleChCaptureTvp7002DisSync.c	- Capture operates in 24-bit RGB discrete sync mode - Capture from VIP0 PortA (via TVP7002, expects 1080i60 source) - CSC converts RGB to YUV - Capture Output 1: Unscaled YUV422SP output - SC - (SC5) - Takes Capture Output 1 as input and outputs 1:1 SC YUV422I output - Display, 1080p60 - Shows YUV422I output from SC	

TI816x TI814x	VC	7: Single CH Capture + SC + Display (1x TVP7002 16b + 1x SII9135 16b, 1080i60+1080P60, YUV420SP)	chains_singleChCaptureSii9135Tvp7002.c	- Dual port capture - VIP0 PortA - TVP7002 in 16-bit YUV422 embedded sync mode, expects 1080i60 source - VIP-SC 1:1 feeds both outputs as below - Capture Output 1: 1920x540 YUV420SP - Capture Output 2: 1920x540 YUV420SP - VIP1 PortA - Sii9135 in 16-bit YUV422 embedded sync mode, expects 1080p60 source - Capture Output 1: Unscaled YUV420SP - Capture Output 2: Scaled 1280x720 YUV420SP - SC (SC5) - Takes the 4 capture outputs as inputs and outputs to a 1920x1080 buffers in a 2x2 formats (we call it SW moasicing) - Display, 1080p60 - Shows YUV422I output from SC	
TI816x TI814x	VC	8: Single CH Capture + SC + Display (1x TVP7002 24b + 1x SII9135 16b, 1080i60+1080P60, YUV420SP)	chains_singleChCaptureSii9135Tvp7002.c	- Same as above except VIP0 PortA operates in 24-bit RGB mode - CSC converts RGB to YUV for this port	

TI816x TI814x	VC	9: Single CH Capture + SC + Display (1x SII9135 16b, 1080P60, YUV422SP)	chains_singleChCaptureSii9135.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 1080p60 source) - Capture Output 1: Unscaled YUV422SP output - SC - (SC5) - Takes Capture Output 1 as input and outputs 1:1 SC YUV422I output - Display, 1080p60 - Shows YUV422I output from SC	
TI816x	VC	a: Single CH Capture + NSF + DEI + Display (1x SII9135 16b, 480i60 -> 480p60 , YUV422I)	chains_singleChCaptureNsfDeiSii9135_480i.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 1080i60 source) - Capture Output 1: Unscaled YUV422SP output - DEI takes Capture output and converts it to YUV422 Progressive - DEI - (DEI_H/DEI) + (SC_H/SC) - 1080i -> 1080p - Display, 1080p60 - Shows YUV422I output from DEI	

TI816x	VC	b: Single CH Capture + NSF + DEI + Display (1x SII9135 16b, 1080i60 -> 1080p60, YUV422I)	chains_singleChCaptureNsfDeiSii9135_1080i.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 1080i60 source) - Capture Output 1: Unscaled YUV422I output - NSF takes capture output and converts it to YUV420 - DEI takes NSF output and converts it to YUV422 - DEI - (DEI_H/DEI) - 1080i -> 1080p - Display, 1080p60 - Shows YUV422I output from DEI	

TI816x	VC	c: Single CH Capture + NSF + DEI + Display (1x SII9135 16b, 480i60 -> 1080p60, YUV422I)	chains_singleChCaptureNsfDeiSii9135_480i_fullscreen.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 480i60 source) - Capture Output 1: Unscaled YUV422I output - NSF takes capture output and converts it to YUV420 - DEI takes NSF output and converts it to YUV422 - DEI - (DEI_H/DEI) + (SC_H/SC) - 480i -> 1080p - Display, 1080p60 - Shows YUV422I output from DEI	

TI816x	VC	d: Single CH Capture + DEI + Display (1x SII9135 16b, 480i60 -> 480p60 , YUV422I)	chains_singleChCaptureNsfDeiSii9135_480i.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 480i60 source) - Capture Output 1: Unscaled YUV422SP output - DEI takes Capture output and converts it to YUV422 Progressive - DEI - (DEI_H/DEI) + (SC_H/SC) - 480i -> 480p - Display, 1080p60 - Shows YUV422I output from DEI	
TI816x	VC	e: Single CH Capture + DEI + Display (1x SII9135 16b, 1080i60 -> 1080p60, YUV422I)	chains_singleChCaptureNsfDeiSii9135_1080i.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 1080i60 source) - Capture Output 1: Unscaled YUV422SP output - DEI takes Capture output and converts it to YUV422 Progressive - DEI - (DEI_H/DEI) + (SC_H/SC) - 1080i -> 1080p - Display, 1080p60 - Shows YUV422I output from DEI	

TI816x	VC	f: Single CH Capture + DEI + Display (1x SII9135 16b, 480i60 -> 1080p60, YUV422I)	chains_singleChCaptureNsfDeiSii9135_480i_fullscreen.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP1 PortA (via Sii9135, expects 480i60 source) - Capture Output 1: Unscaled YUV422SP output - DEI takes Capture output and converts it to YUV422 Progressive - DEI - (DEI_H/DEI) + (SC_H/SC) - 480i -> 1080p - Display, 1080p60 - Shows YUV422I output from DEI	

TI816x	VC	g: Single CH Capture + SC + Display (1x TVP7002 16b, 1080i60, YUV420SP, FieldsMerged -> 1080p30, YUV422I)	chains_singleChCaptureTvp7002FieldMerged.c	- Capture operates in 16-bit YUV422 embedded sync mode - Capture from VIP0 PortA (via TVP7002, expects 1080i60 source) - Capture Output 1: Unscaled YUV420SP output, field merged - Capture Output 2: Unscaled YUV420SP output, field merged - SC: SC5 driver, Takes Capture Output 1,2 alternatively as input and outputs 1:1 SC YUV422I output - Display: 1080p30 - Shows YUV422I output from SC	

Link Details

The links which are used to create all the above mentioned chains are described below.

1. **Capture**: This single link is used to capture frames from all the four VIP parser ports. It has no input queue and 4 output queues to support 4 instances of VIP ports. It can capture a maximum of 16 channels from 4 VIP ports. IOCTL CAPTURE_LINK_CMD_DETECT_VIDEO is supported for this link which is used to detect the video and fetch the image properties from the sensor. After capturing, this link can either send all the captured channels to 1 output queue or it can distribute the channels among the 4 output queues. For e.g., in dual 1080p mode, it can capture one channel each from two VIP ports, configure the VIP in dual output mode and send the 4 (2 VIP ports * 2 outputs) outputs to 4 output queues.
2. **Noise filter**: This link has one input and two output queues. It takes multiple frames from different channels as input and gives the same number of frames/channels as outputs after noise-filtering. Output can be split in two output queues depending on a configuration switch which allows either using output queue 0 for all the channels or using both the output queues for half of the total available channels.
3. **DEI**: This link has one input and two output queues. It takes multiple frames from different channels as input and gives the same number of frames/channels as outputs after de-interlacing. DEI always gives two outputs: a) de-interlaced non-scaled output, which goes to VIP scalar, and b) de-interlaced scaled output. First output is in YUV420 format whereas the second output is in YUV422 format hence it can be given directly to the display. Each of the outputs can be enabled/disabled individually for this link.

4. **Scalar - Software Mosaic**: This link has one input and one output queue. It takes multiple frames from different channels as input and gives one channel as output after processing. This single channel output has all the up-scaled/downscaled frames, arranged in a mosaic format. Scaling is done after every n msec which is configurable. This link supports SCALAR_SW_MS_LINK_CMD_SWITCH_LAYOUT IOCTL which is used to switch layout for the number of windows/channels chosen for display. The starting channel number for the layout can also be changed from the application. Channels which are not used for mosaic are returned immediately to the previous link.
5. **Scalar**: This link has one input and one output queue. It takes multiple frames from different channels as input and gives the same number of frames/channels as outputs after scaling.
6. **Display**: This link has only one input queue and no output queue. It takes multiple channels as input and shows only one pre-selected channel on the connected display. The channel which will be displayed can be changed by using DISPLAY_LINK_CMD_SWITCH_CH IOCTL. All the other channels are returned immediately to the previous link.
7. **Display – Hardware Mosaic**: It has two input queues and no output queue. It takes multiple channels as inputs and displays the selected subset of channels in mosaic format on the connected display. All other channels are returned immediately to the previous link. The two input queues can be used to connect to two different links like DEI and DEI_H. In this case, both input queues will receive frames from the previous links. Depending on the display link settings, it will choose a subset from all the available channels, create a mosaic display out of them and display it accordingly.
8. **Graphics**: It is a pseudo-link for the graphics module which initializes the graphics driver and displays a pre-defined logo on the connected output. It is meant only for the system-level integration example.
9. **Null**: It is used as a dummy sink for links having multiple outputs (like DEI) in which one output is not used and hence needs to be returned immediately. It can be used for debugging purpose as well. For e.g. if display link doesn't work as expected, the previous link in the chain can be connected to the NULL link to make sure that other links in the chain are working properly.

The interfaces files and the corresponding source code for all the above links can be found in *packages\ti\psp\examples\common\vps\chains\links* folder.

S No	Link	Interface file	Source code (folder)
1	Capture	captureLink.h	capture
2	Noise filter	nsfLink.h	nsf
3	De-interlacer	deiLink.h	dei
4	Scalar Software Mosaic	scalarSwMsLink.h	scalarSwMs
5	Scalar	scLink.h	scalar
6	Display	displayLink.h	display
7	Display Hardware Mosaic	displayHwMsLink.h	displayHwMs
8	Graphics	grpxLink.h	grpx
9	Null	nullLink.h	null

TI81xx-HDVPSS MultiCore Arch

HDVPSS Software MultiCore Architecture

Need

HDVPSS driver is running on Ducati M3 (hosting BIOS6) and applications running on Ducati M3 can interface via FVID2, If we required to control HDVPSS from other cores, we would require to re-write/port HDVPSS drivers to other cores and/or operating system. Which requires significant effort to develop and maintain drivers in multiple operating system (and CPUs). Additionally resource allocation/management across cores could be challenging.

Overview

The basic idea is to have HDVPSS driver running on Ducati M3 along with a daemon (referred as Proxy Server) and other cores will run an dummy drivers (referred as clients) that would request host to perform required operations, based on the request from the local applications.

The interface exposed by client could be same as FVID2, or any other interface (as required by architecture of the local applications/OS)

Example

1. Assuming that we would require to access display driver from A8 running Linux. The client running on A8, would present itself as the display driver to Linux (V4L2 Display Driver or FBDev Display Driver), translate Linux display requests to FVID2 commands and have them processed by Proxy Server.
2. Assuming that we are running a client on BIOS6 on DSP and we have applications running on DSP that use FVID interfaces. the client would simply transfer the FVID2 call to the proxy server and return back the response to application that made the call.

Dependencies

IPC

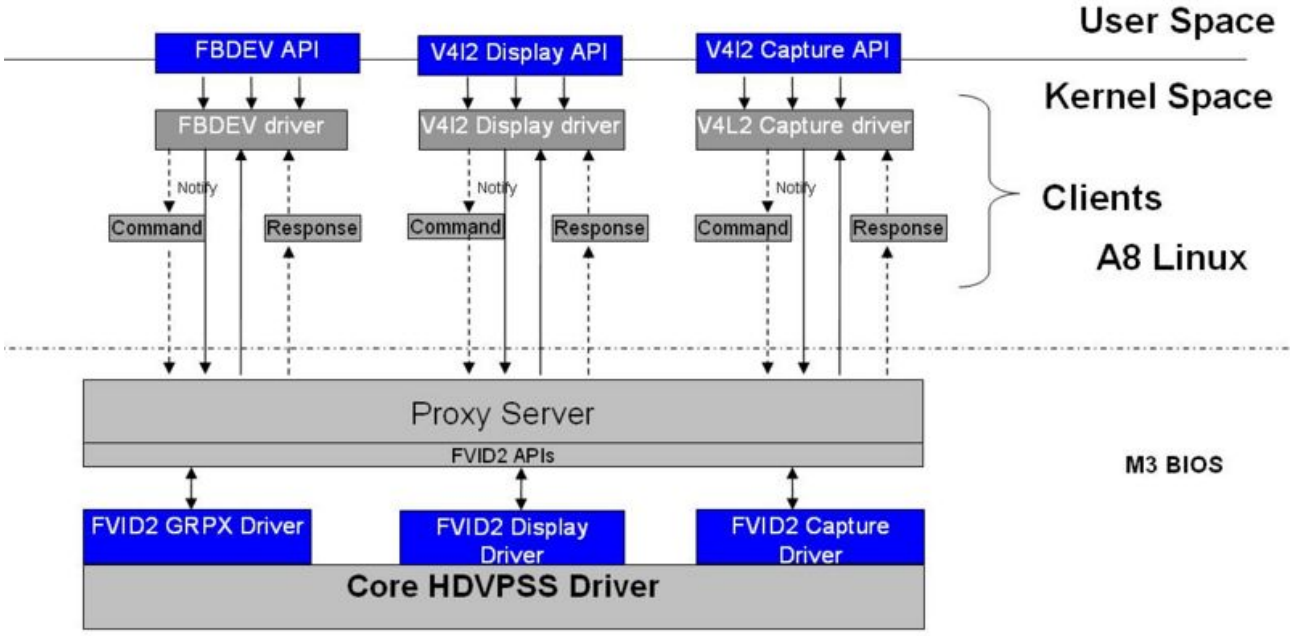
Proxy Server relies on Texas Instruments IPC for inter-core communication in particular on Notify module of IPC.

Customized FVID2 Interace

Proxy Server Interface build upon FIVD2 interfaces, Please refer Interface Sections for details...

Protocol

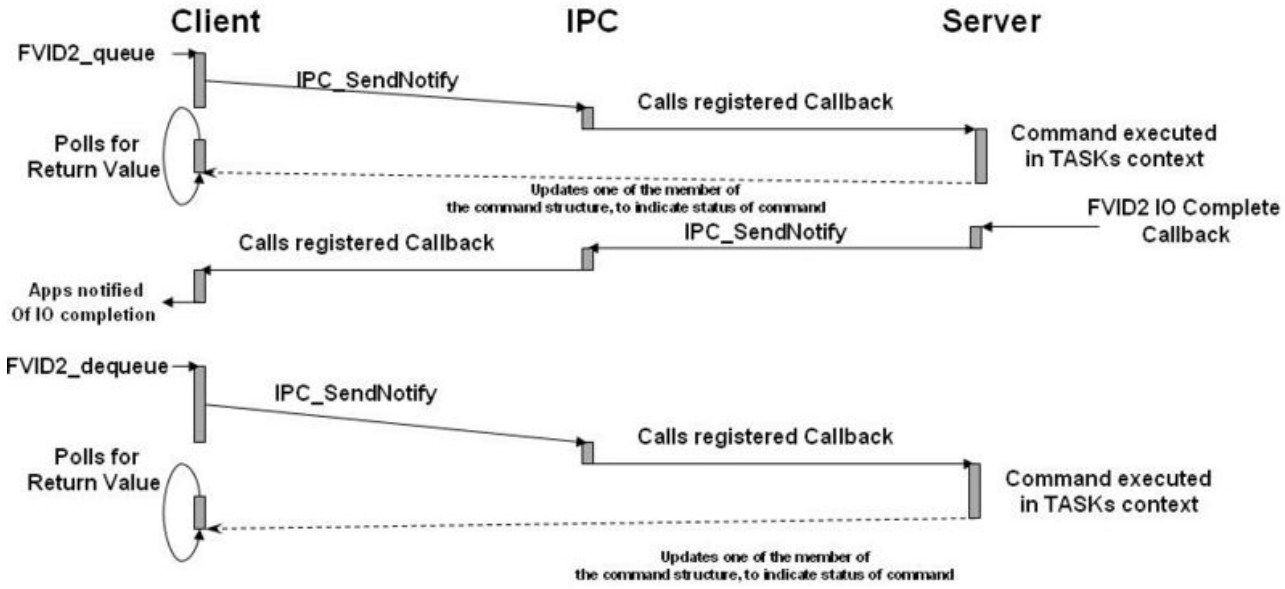
Proxy Server is based on command response mechanism. The following pictures depicts the same for display/capture/FBDev driver running on A8 Linux.



Protocol

Sample Queue and DeQueue operation

Below pictures depicts the sequence of operations done on host and client to process an command from remote core.



Queue De-queue

Server

Reserved Notify

Control commands such as FVID2_init, FVID2_create, FVID2_delete, FVID2_deInit and display controller command, will require to use a reserved IPC notify number. (An initialization time parameter)

Notify event number to be used by client for IO

On successful creation of a FVID2 stream, an notify event number would be allocated. Clients are expected to use this for all further IO on this stream

Task that process commands from clients

There would be 5 tasks

1 Control task, used to execute control commands

1 Task each to handle streams (Capture, Display, Memory2Memory and Graphics)

Priorities of these tasks is initialization time parameter. Could be used for load balancing

Client requires to identify the class of stream during channel creation. (Left to clients to enable load balancing

A separate Q for each task

Depth of the Q is initialization time parameter

Requires Physical address of the command Structure

Clients

Implements the functions defined by fvid2.h

Essentially a thin driver that would talk to host to perform fvid2 operations

Structure of each command is predefined, clients require to order any request into this

All control commands are requested via reserved notification event number

A callback requires to be registered with IPC.Notify on the allocated notification event number. This callback should handle completed IO requests

Interface

Proxy Server build upon FIVD2 interface, where each FVID2 API is defined as command structure with 3 additional parameters...

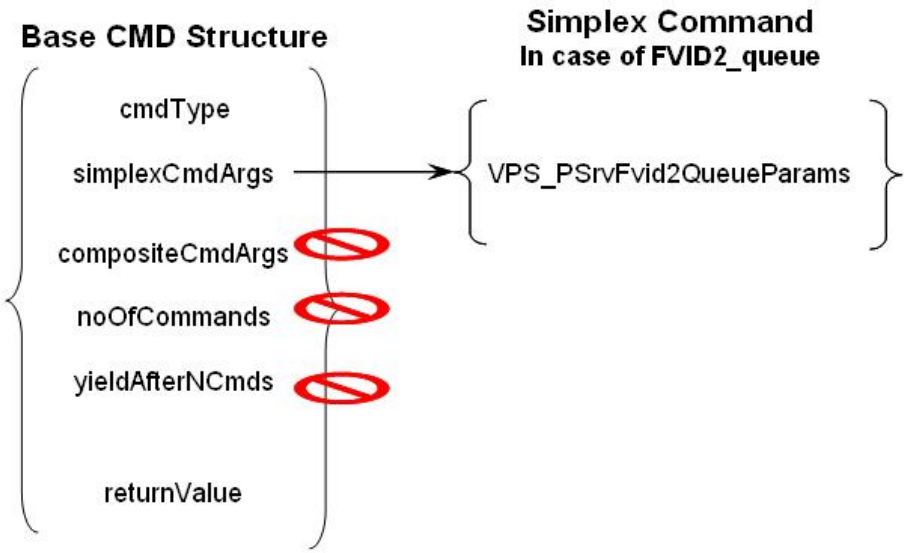
[IN] VPS_PSrvCommands command;

[IN] UInt32 reserved;

[OUT] Int32 returnValue;

Where *command* identifies type of FVID2 API and *returnValue* to hold the return value returned by HDVPSS drivers.

Each FVID2 command is encapsulated in a command structure defined below



Composite Command is not supported

Proxy Server Features

1. Single API to initialize Proxy Server
2. Configurable to serve predefined cores
3. Priorities of tasks that process clients commands
4. Number of events for a given core, each core could potentially have different number of events and different event numbers...
5. Configurable reserved notify event number. Applies to all cores.

Sample Application

This release provides a application to initialize proxy server. Linux FBDEV driver for TI81xx devices, uses this application to control, configure and display images from a different core.

Article Sources and Contributors

TI81xx-HDVPSS-UserGuide *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=130605 *Contributors*: A0868651, Anuj.aggarwal, HardikShah, Jadavbrijesh, MugdhaKamoolkar, PurushotamKumar, SivarajR, Sudhir, SujithShivalingappa

T181xx-HDVPSS Overview *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129171 *Contributors*: Anuj.aggarwal, BrijeshJadav, HardikShah, Jadavbrijesh, SivarajR

T1816X-HDVPSS-HW Overview *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=68887 *Contributors*: Anuj.aggarwal

T1814X-HDVPSS-HW Overview *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129177 *Contributors*: Anuj.aggarwal, Jadavbrijesh, SujithShivalingappa

UserGuideHdvpssFolderOrg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129122 *Contributors*: Jadavbrijesh, SujithShivalingappa

UserGuideFVID2 *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=125959 *Contributors*: Anuj.aggarwal, HardikShah, Vikramgara

UserGuideHdvpssPlatformAPIs *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129180 *Contributors*: HardikShah, Jadavbrijesh

UserGuideHdvpssDisplayDriver *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129206 *Contributors*: A0868651, Anuj.aggarwal, BrijeshJadav, HardikShah, Jadavbrijesh, Lkreddy, SivarajR, Vikramgara

UserGuideHdvpssM2mDriver *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129200 *Contributors*: A0131716, Anuj.aggarwal, HardikShah, Jadavbrijesh, Lkreddy, MugdhaKamoolkar, SivarajR, SujithShivalingappa, Vikramgara

UserGuideHdvpssTi816xDeiM2mDriver *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=122624 *Contributors*: Anuj.aggarwal, MugdhaKamoolkar, SivarajR

UserGuideHdvpssTi814xDeiM2mDriver *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129198 *Contributors*: A0131716, Anuj.aggarwal, Jadavbrijesh, SivarajR, SujithShivalingappa

UserGuideHdvpssCaptureDriver *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129203 *Contributors*: Anuj.aggarwal, HardikShah, Jadavbrijesh, Kedarsc, MugdhaKamoolkar, SivarajR, SujithShivalingappa

TI81xx-external video drivers *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=72409 *Contributors*: BrijeshJadav, HardikShah, Kedarsc, SivarajR

UserGuideHdvpssIntegExample *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=129209 *Contributors*: A0131716, Anuj.aggarwal, Jadavbrijesh, Kedarsc, SivarajR, SujithShivalingappa, Vikramgara

TI81xx-HDVPSS MultiCore Arch *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=79998 *Contributors*: HardikShah, SivarajR, SujithShivalingappa

Image Sources, Licenses and Contributors

Image:TIBanner.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TIBanner.png *License*: unknown *Contributors*: Sriram

Image:Install_1.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_1.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Install_2.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_2.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Install_3.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_3.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Install_4.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_4.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Install_5.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_5.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Install_6.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_6.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Install_8.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_8.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Install_9.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_9.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Install_10.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Install_10.PNG *License*: unknown *Contributors*: Anuj.aggarwal

Image:Netra DSS Display.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Netra_DSS_Display.png *License*: unknown *Contributors*: PurushotamKumar

Image:Ti816x-HDVPSS.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Ti816x-HDVPSS.jpg *License*: unknown *Contributors*: Anuj.aggarwal, SivarajR

Image:Ti814x-HDVPSS_updated.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Ti814x-HDVPSS_updated.jpg *License*: unknown *Contributors*: SujithShivalingappa

Image:TI8107-HDVPSS Hardware.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TI8107-HDVPSS_Hardware.jpg *License*: unknown *Contributors*: Jadavbrijesh

File:TopLevel.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TopLevel.PNG *License*: unknown *Contributors*: SujithShivalingappa

File:BuildFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BuildFolder.PNG *License*: unknown *Contributors*: SujithShivalingappa

File:DocsFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DocsFolder.PNG *License*: unknown *Contributors*: SujithShivalingappa

File:MakeFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:MakeFolder.PNG *License*: unknown *Contributors*: SujithShivalingappa

File:PsVpsFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:PsVpsFolder.PNG *License*: unknown *Contributors*: SujithShivalingappa

File:EgOthersFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:EgOthersFolder.PNG *License*: unknown *Contributors*: Jadavbrijesh, SujithShivalingappa

File:EgFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:EgFolder.PNG *License*: unknown *Contributors*: SujithShivalingappa

File:I2cplatformFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:I2cplatformFolder.PNG *License*: unknown *Contributors*: SujithShivalingappa

File:DriversFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DriversFolder.PNG *License*: unknown *Contributors*: SujithShivalingappa

Image:YUV420_semiplanar_changed.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:YUV420_semiplanar_changed.jpeg *License*: unknown *Contributors*: HardikShah

Image:YUV422_interleaved.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:YUV422_interleaved.jpeg *License*: unknown *Contributors*: HardikShah

Image:RGB888_packed.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:RGB888_packed.jpeg *License*: unknown *Contributors*: HardikShah

Image:YUV420 addr.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:YUV420_addr.jpeg *License*: unknown *Contributors*: HardikShah

Image:YUV422_addr.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:YUV422_addr.jpeg *License*: unknown *Contributors*: HardikShah

Image:FVID_FrameListCapture.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:FVID_FrameListCapture.PNG *License*: unknown *Contributors*: HardikShah

Image:FVID_FrameListDisplay.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:FVID_FrameListDisplay.PNG *License*: unknown *Contributors*: HardikShah

Image:FVID_ProcessList.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:FVID_ProcessList.PNG *License*: unknown *Contributors*: HardikShah

Image:FVID2_FrameList.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:FVID2_FrameList.png *License*: unknown *Contributors*: HardikShah

Image:FVID2_Queue_DeQueue_displayBig.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:FVID2_Queue_DeQueue_displayBig.PNG *License*: unknown *Contributors*: HardikShah

Image:FVID2_Queue_DeQueue_capture.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:FVID2_Queue_DeQueue_capture.PNG *License*: unknown *Contributors*: HardikShah

Image:FVID2_ProcessList.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:FVID2_ProcessList.PNG *License*: unknown *Contributors*: HardikShah

Image:ProcessList_Queue_DeQueue.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:ProcessList_Queue_DeQueue.PNG *License*: unknown *Contributors*: HardikShah

Image:DisplayFlowChart.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DisplayFlowChart.png *License*: unknown *Contributors*: Anuj.aggarwal

Image:DctrlTI8107.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DctrlTI8107.JPG *License*: unknown *Contributors*: Jadavbrijesh

Image:DisplayCtrlCentaurusMinimal.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DisplayCtrlCentaurusMinimal.jpg *License*: unknown *Contributors*: Anuj.aggarwal

Image:DisplayCtrlNetraMinimal.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DisplayCtrlNetraMinimal.JPG *License*: unknown *Contributors*: Anuj.aggarwal, SivarajR

Image:DCTRL_Topology_TI814x.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DCTRL_Topology_TI814x.JPG *License*: unknown *Contributors*: Anuj.aggarwal

Image:DCTRL-Topology-TI8107.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DCTRL-Topology-TI8107.jpg *License*: unknown *Contributors*: Jadavbrijesh

Image:DCTRL_Topology.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DCTRL_Topology.JPG *License*: unknown *Contributors*: SivarajR

Image:PpDisplayPathMinimal.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:PpDisplayPathMinimal.jpg *License*: unknown *Contributors*: Anuj.aggarwal, SivarajR

Image:SecDisplayPathMinimal.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SecDisplayPathMinimal.JPG *License*: unknown *Contributors*: Anuj.aggarwal, SivarajR

Image:GrpxPathMinimal.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:GrpxPathMinimal.jpg *License*: unknown *Contributors*: Anuj.aggarwal, SivarajR

Image:Nsf-m2m-path.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Nsf-m2m-path.JPG *License*: unknown *Contributors*: SivarajR

Image:SEC0_SC5.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SEC0_SC5.JPG *License*: unknown *Contributors*: Lkreddy, SivarajR

Image:BP01_SC5_M2M.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BP01_SC5_M2M.JPG *License*: unknown *Contributors*: Lkreddy, SivarajR

Image:HDVPSS_SEC01_SC34_VIP01.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:HDVPSS_SEC01_SC34_VIP01.JPG *License*: unknown *Contributors*: SivarajR

Image:M2M FlowChart.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:M2M_FlowChart.png *License*: unknown *Contributors*: Anuj.aggarwal

Image:Image Properties 720X480.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Image_Properties_720X480.png *License*: unknown *Contributors*: Anuj.aggarwal

Image:DeihSingleScM2MPath.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DeihSingleScM2MPath.JPG *License*: unknown *Contributors*: SivarajR

Image:DeiSingleScM2MPath.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DeiSingleScM2MPath.JPG *License*: unknown *Contributors*: SivarajR

Image:DeihSingleScVipM2MPath.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DeihSingleScVipM2MPath.JPG *License*: unknown *Contributors*: SivarajR

Image:DeiSingleScVipM2MPath.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DeiSingleScVipM2MPath.JPG *License*: unknown *Contributors*: SivarajR

Image:DeihDualScaleM2MPath.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DeihDualScaleM2MPath.JPG *License*: unknown *Contributors*: SivarajR

Image:DeiDualScaleM2MPath.JPG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DeiDualScaleM2MPath.JPG *License*: unknown *Contributors*: SivarajR

Image:DeiHqMqMode1M2MPath.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DeiHqMqMode1M2MPath.png *License*: unknown *Contributors*: Anuj.aggarwal, SivarajR

Image:DeiHqMqMode1Exmpl.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DeiHqMqMode1Exmpl.png *License*: unknown *Contributors*: Anuj.aggarwal

Image: TI814x_DEI_SC1_WB0.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TI814x_DEI_SC1_WB0.jpeg *License*: unknown *Contributors*: SujithShivalingappa

Image: TI814x_DEI_VIP0_SC3.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TI814x_DEI_VIP0_SC3.jpeg *License*: unknown *Contributors*: SujithShivalingappa

Image: TI814x_DEI_SC1_VIP0_SC3.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TI814x_DEI_SC1_VIP0_SC3.jpeg *License*: unknown *Contributors*: SujithShivalingappa

Image: TI814X_SC2_WB1.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TI814X_SC2_WB1.jpg *License*: unknown *Contributors*: A0131716

Image: TI814X_SC4_VIP1.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TI814X_SC4_VIP1.jpg *License*: unknown *Contributors*: A0131716

Image: TI814X_SC2_SC4_VIP1_WB1.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TI814X_SC2_SC4_VIP1_WB1.jpg *License*: unknown *Contributors*: A0131716

Image:Vip-capture-paths.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Vip-capture-paths.png *License*: unknown *Contributors*: Anuj.aggarwal, SivarajR

Image:Chain1.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Chain1.jpeg *License*: unknown *Contributors*: Anuj.aggarwal

Image:Chain3.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Chain3.jpeg *License*: unknown *Contributors*: Anuj.aggarwal

Image:Chain4.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Chain4.jpeg *License*: unknown *Contributors*: Anuj.aggarwal

Image:Chain5.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Chain5.jpeg *License*: unknown *Contributors*: Anuj.aggarwal

Image:Chain6.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Chain6.jpeg *License*: unknown *Contributors*: Anuj.aggarwal

Image:Protocol.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Protocol.jpg *License*: unknown *Contributors*: SujithShivalingappa

Image:Q-Dq.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Q-Dq.jpg *License*: unknown *Contributors*: SujithShivalingappa

Image:simplexCommand.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:SimplexCommand.jpg *License*: unknown *Contributors*: SujithShivalingappa