BSP
User Guide

Contents

Articles

BSP UserGuide 1

UserGuideBSPFolderOrg 37

Vayu-Overview 44

TDA3XX-Overview 50

BSP-Software Overview 55

UserGuideFVID2 Vayu 58

UserGuideBSPPlatformAPIs 79

UserGuideBSPCaptureDriver 85

UserGuideBSPM2mVpeDriver 106

UserGuideBspDisplayDriver 116

References

Article Sources and Contributors 129

Image Sources, Licenses and Contributors 130

BSP UserGuide

About this Manual

This document describes how to install and work with the Texas Instruments BSP drivers on TDA2XX EVM , TDA3XX EVM and TI814x EVM. The BSP package serves to provide a fundamental software platform for development, deployment and execution of video applications. BSP 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 BSP package
- Build the BSP package

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

- BSP introduction
- BSP Capture (VIP) drivers
- BSP ISS capture drivers on TDA3xx
- BSP Video Processing Engine (VPE Memory to Memory) drivers
- BSP ISS Memory to memory drivers on TDA3xx
- BSP Display sub-system drivers
- BSP Serial Drivers - I2C, UART, MCSPI, MCASP

Getting Started

System Requirements

Refer to 'Dependencies' section of release notes.

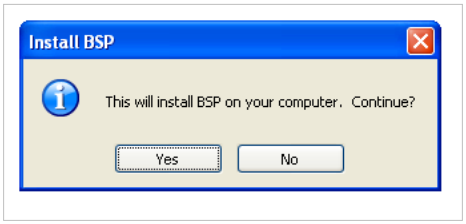
Installation

BSP 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.

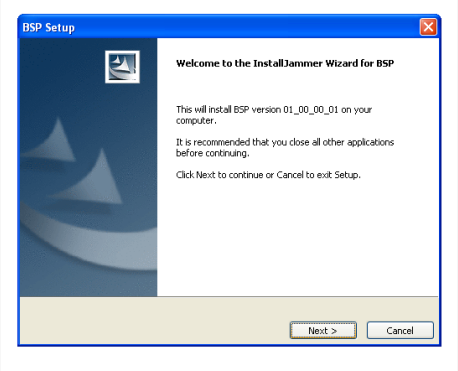
Following are the installation steps:

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

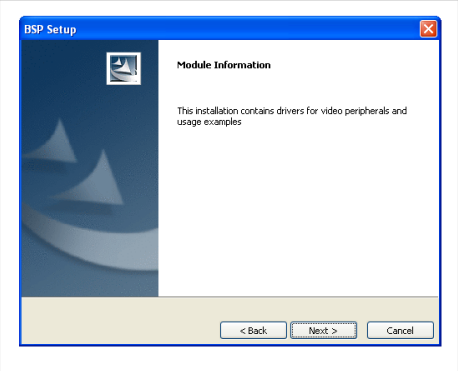
- Dialogue box appears to confirm the package installation click Yes to install.



- Welcome screen appears. Click next to continue.

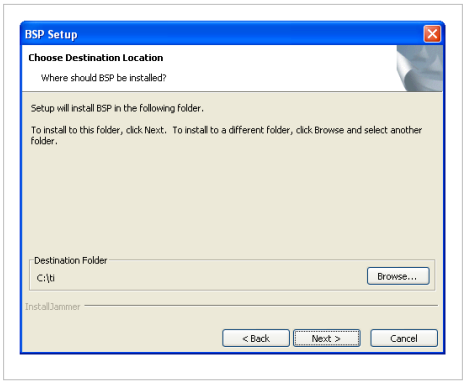


- Module information appears for the installation package. Click next to continue.

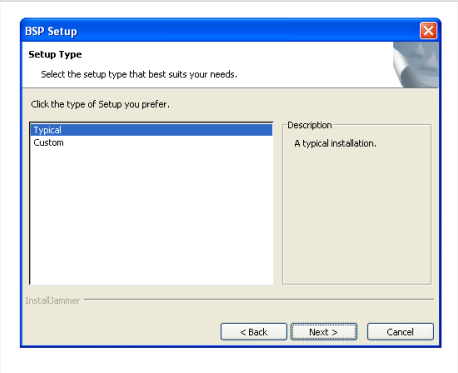


- License agreement appears. Accept the terms of agreement after reading and click yes to continue.

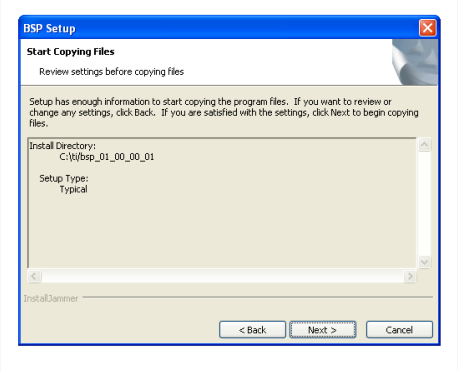
- Destination folder screen appears. Select the installation folder where other TI tools like CCS, BIOS are installed.



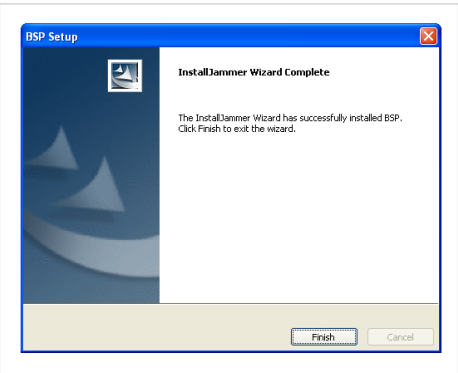
- Select Typical type of installation.



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



- Click on Finish to complete the installation



This completes the installation of the package.

Running BSP Application on TDA3XX EVM

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

Common Steps for all examples

- Installing CCS
  - CCSv5.4.0 Release Candidate 2 (Build 91) at http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v5
  - TDA3xx Chip Support Package: device xml files, GEL files, etc. for CCS configuration https://spsemea.itg.ti.com/sites/cmss_Platforms/ADAS_LOW/Shared%20Documents/Forms/AllItems.aspx?RootFolder=%2fsites%2fcmss_Platforms%2fADAS_LOW%2fShared%20Documents%2fPG1%2e0%20Silicon%20Wak FolderCTID=&View=%7bC8D3F3DD-41BB-4AD3-B4A0-6B0B0A57BB38%7d
- Create a new Target Configuration file with device as TDA3XX and specify the Emulator type which you are using.
- Once Tda3xx is launched with the target configuration created , then load all the Gel files required.
- Switch settings in TDA3XX board should set to SYSBOOT[0-7] - "00100000" and SYSBOOT[8-15] - "00000000".
- Connect to M4_IPU1 core 0
- Load and Run the respective examples
- In case of ADV7611 HDMI Receiver capture through VIP, SW8000[1:4] should be set to 1010. EDID programming to the EEPROM is not required as the ADV7611 driver uses the internal EDID RAM
- In case of Digital parallel capture from OV10640 and AR0132 sensors through ISS on TDA3xx, make sure that the CPLD is programmed with the firmware revision REVB1. The binary for this firmware revision can be found in pspdrivers_\docs\tda3xx\cpld\
- If I2C Read/Write fails for OV10640 sensor, change the I2C frequency to 100KHz in gBoardTda3xxI2cInstData[1].busClkKHz in the file bspdrivers_\src\board\src\bsp_boardTda3xx.c file and rebuild ISS examples.

```
diff --git a/bspdrivers_/src/boards/src/bsp_boardTda3xx.c
b/bspdrivers_/src/boarindex 517306b..ba76593 100755
--- a/bspdrivers_/src/boards/src/bsp_boardTda3xx.c
+++ b/bspdrivers_/src/boards/src/bsp_boardTda3xx.c
@@ -281,7 +281,7 @@ static Bsp_BoardI2cInstData
gBoardTda3xxI2cInstData[] =
        BSP_DEVICE_I2C_INST_ID_1,           /* instId */
        SOC_I2C2_BASE,                      /* baseAddr */
        CSL_INTC_EVENTID_I2CINT2,           /* intNum */
-       400U                                /* busClkKHz */
+       100U                                /* busClkKHz */
    }
};
```

- For VIP capture from Multi-deserializer board, the multi-deserializer board should be be configured for 4-channel operation. The CN2, CN3 and CN4 jumper settings should be set as per below picture

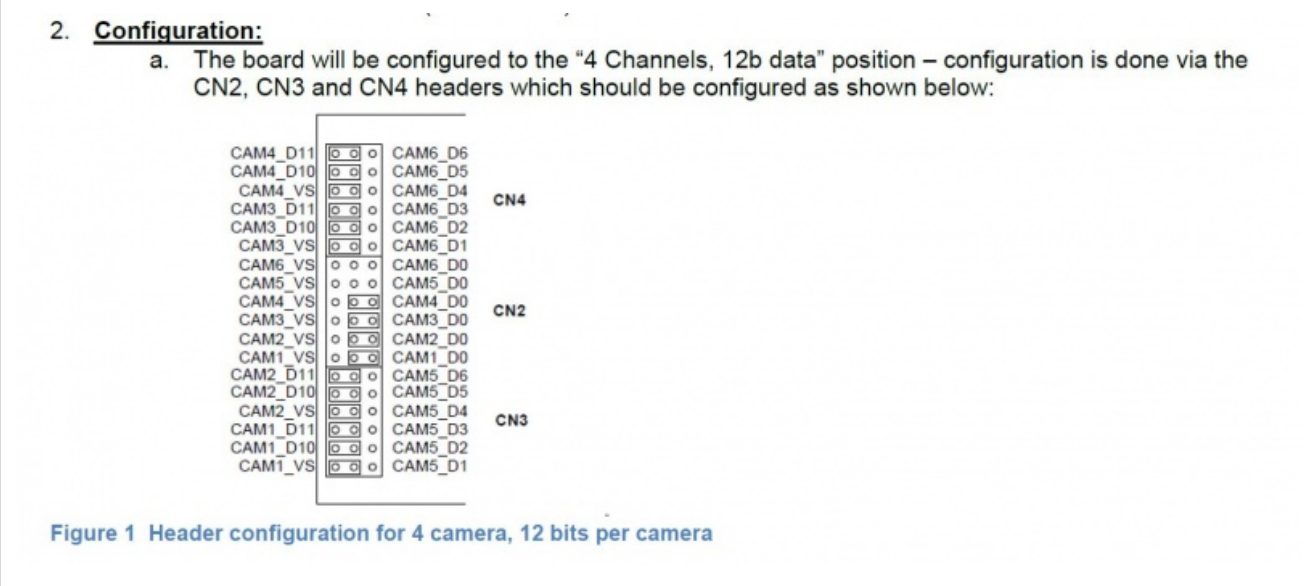


Figure 1 Header configuration for 4 camera, 12 bits per camera

- In case of Multi-deserializer capture through VIP or ISS capture from sensors, the following board modification is required in the base board to avoid I2C issues

ER15: ADJUST THE I2C PULLUPS TO PULLUPS TO IMPROVE OPERATION WITH
FROM 101370-2202R TO 101370-1003R, 2.2K TO 10K
SHEET 8, R9315, R9316
SHEET 9, R9371, R9372, R9373, R9374
SHEET 28, R9391, R9392, R9394, R9395
SHEET 29, R9397, R9398
SHEET 30, R1058, R1059

- For OV10640 Parallel input, change the IO voltage to 1.8V from 3.3V by removing resistor RJ26 from pins 1-2 and installing it on pins 2-3.

Running Capture example

- Make sure OV sensor OV1063x is connected to the base EVM before running the example.
- Before running any ISS capture examples, ensure required / supported sensor is connected. http://ap-fpdsp-swapps.dal.design.ti.com/index.php/BSP_UserGuide#Connecting_Sensors_To_EVM

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option.
- Save the buffer with the command mentioned in the log and verify the buffer.

Constraints:

- For running on EVM, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda3xx-evm**

Running Display example

Configuring Display:

- Make sure the LCD panel is connected to the base EVM before running the example binary.
- Make sure the HDMI is connected to TV from the EVM for HDMI display options.
- Make sure the SD Cable is connected to TV from the EVM for SDDAC options.

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option and load the buffer as suggested.
- Content will be displayed on the LCD panel connected to EVM for options till 3
- Content will be displayed on the HDMI TV connected to EVM for options 4, 5 and 6
- Content will be displayed on the SD TV connected to EVM for options 7 and 8.

Constraints:

- For running on EVM, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda3xx-evm**

Note:

- Test Input files for Display Sample application will be in "bspdrivers_\docs\test_inputs\DisplayInput.rar"
- For Option 0 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf"
- For Option 1 : Load the input file "Logo_rgb888_prog_packed_1920_1080.tigf"
- For Option 2 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf" for YUV422I_YUYV frames and "Logo_rgb888_prog_packed_1920_1080.tigf" for BGR24_888 frames
- For Option 3 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf" for YUV422I_YUYV frames and "Logo_rgb888_prog_packed_1920_1080.tigf" for BGR24_888 frames
- For Option 4 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf"
- For Option 5 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf"
- For Option 6 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf"

Running Loopback example

- Make sure OV sensor and LCD Panel is connected to to the Base EVM before running the binary.

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option.
- Captured Content will be displayed on the LCD panel connected to EVM

Constraints:

- For running on EVM, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda3xx-evm**

Running I2C LED Blink example

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application
- LEDs on Tda3xx CPU Board will be blinking

Build:

- For running on EVM, select **PLATFORM := tda3xx-evm** while building the binary

Running UART example

- Make sure mini USB Cable is connected to host PC and the EVM, configure a serial terminal like Teraterm for standard baud rate(115200) and connect on newly installed Port.

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option.
- Input a text of 1k bytes for RX, you will see the text characters on terminal for TX

Build:

- For running on EVM, select **PLATFORM := tda3xx-evm** while building the binary

Running McSPI example

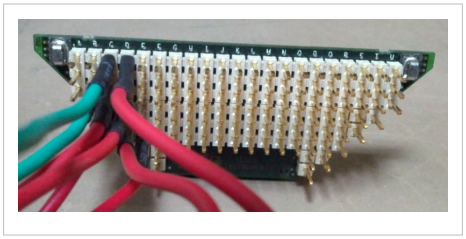
- For spi1tospi2 example Make sure the pins of SPI1 and SPI2 instances on the EVM are blue wired and board is configured for TX/RX Connections.

For TDA3xx please find the images below for connections. SPI pins to be connected to Visibility Connector J28 present beside SOC.

Please note that as UART3 pin is muxed with SPI1 SCLK for TDA3xx, UART1 terminal must be connected to give user input.

SPI1 to SPI2

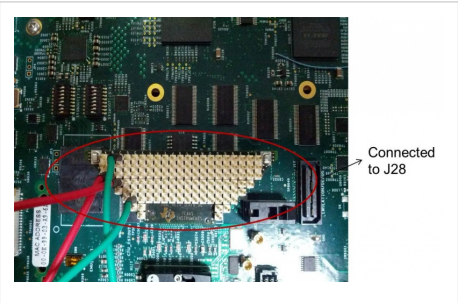
SPI1	SPI2
SPI1 CS0 (Pin D4)	SPI2 CS0 (Pin D1)
SPI1 SCLK (Pin D5)	SPI2 SCLK (Pin C3)
SPI1 MOSI (Pin B3)	SPI2 MISO (Pin C1)
SPI1 MISO (Pin C4)	SPI2 MOSI (Pin E6)



- For loopback example Make sure the Tx and Rx pins of the SPI instance to be tested are connected on the EVM and board is configured for TX/RX Connections. Loopback test is supported on SPI1 and SPI2 instances.

SPI LOOPBACK

SPI1 MOSI (Pin B3)	SPI1 MISO (Pin C4)
SPI2 MISO (Pin E6)	SPI2 MOSI (Pin C1)



CCS connection:

- Load the generated Xem4 file in CCS
- Run the application.
- Tx and Rx buffers are verified for data integrity, and the result is printed on console.

Build:

- For running on EVM, select **PLATFORM := tda3xx-evm** while building the binary

Running ISS Capture Example

CCS connection:

- Load the generated bsp_examples_captureIss_m4vpss_release.xem4 file in CCS
- Run the application, Select any of the given option (Ensure the selected sensor is connected to EVM)
- Save the buffer with the command mentioned in the log and verify the buffer

Build:

- For running on EVM, select **PLATFORM := tda3xx-evm** and **PACKAGE_SELECT := all** while building the binary

Note:

- Make sure OV10640 CSI2 sensor connected on CSI2 connector and OV10640 Parallel Sensor on the LI connected, and Aptina AR0132 on aptina connector are connected to the base EVM before running the example. http://ap-fpdsp-swapps.dal.design.ti.com/index.php/BSP_UserGuide#Connecting_Sensors_To_EVM
- **If I2C Read/Write fails for OV10640 sensor, change the I2C frequency to 100KHz in gBoardTda3xxI2cInstData[1].busClkKHz in the file bspdrivers_\src\board\src\bsp_boardTda3xx.c file and rebuild the capture example.**
- Make sure that the CPLD is flashed with the REVB1 firmware. This firmware binary can be found in the pspdrivers_\docs\tda3xx\cpld\ folder.

Running ISS OTF Capture Example

CCS connection:

- Load the generated bsp_examples_captureIssOtf_m4vpss_release.xem4 file in CCS
- Run the application, Select any of the given option (Ensure the selected sensor is connected to EVM)
- Save the buffer with the command mentioned in the log and verify the buffer

Build:

- For running on EVM, select **PLATFORM := tda3xx-evm** and **PACKAGE_SELECT := all** while building the binary

Running ISS Loopback Example

CCS connection:

- Load the generated bsp_examples_loopbackiss_m4vpss_release.xem4 file in CCS
- Run the application, Select any of the given option
- Output is displayed on TV through HDMI interface

Build:

- Building for EVM, select **PLATFORM := tda3xx-evm**, **PACKAGE_SELECT := all** and **INCLUDE_WDR_LDC := yes** while building the binary

Note:

- Make sure OV10640 CSI2 sensor connected on CSI2 connector and OV10640 Parallel Sensor on the LI connected, and Aptina AR0132 on aptina connector are connected to the base EVM before running the example. http://ap-fpdsp-swapps.dal.design.ti.com/index.php/BSP_UserGuide#Connecting_Sensors_To_EVM
- Make sure display output is connected to TV through HDMI interface.
- Make sure that the CPLD is flashed with the REVB1 firmware. This firmware binary can be found in the pspdrivers_\docs\tda3xx\cpld\ folder.
- **If I2C Read/Write operation fails for OV10640 sensor, change the I2C frequency to 100KHz in gBoardTda3xxI2cInstData[1].busClkKHz in the file bspdrivers_\src\board\src\bsp_boardTda3xx.c file and rebuild Loopback examples.**
- Application runs two pass Memory to Memory WDR flow for the frames captured from OV10640 sensor and it runs single pass Memory to Memory ISP for the frames captured from AR0132 sensor
- **ISP and Sensor Parameters are not tuned for all lighting conditions, it works with the fixed set of parameters, which may not give good quality output in difference lighting condition.**
- **For using this example, install add on package for ISS on starterware. Details can be found in starterware releasenotes/userguide. After installing starterware ISS add on package, please make sure that LDC and WDR files vpshal_issglbce.c and vpshal_issldc.c are present in the folder starterware_\vpslib\hal\src**
- **A board power cycle is required after running any given option. Without doing power cycle, I2C write fails**

Running ISS Memory to memory Example

CCS connection:

- Load the generated bsp_examples_m2mIss_m4vpss_release.xem4 file in CCS
- Run the application
- Load any 1280x720 size Bayer RAW image with the format GRBG at the given location using CCS scripting console.
- It generates 8 output files, save the buffer with the command mentioned in the log and verify the buffer.

Build:

- For running on EVM, select **PLATFORM := tda3xx-evm** and **PACKAGE_SELECT := all** while building the binary

Note:

- Sample Reference Input files for ISS M2M Sample application is present at "bspdrivers_\docs\test_inputs\ISSM2M_BayerRAW_GRBG_1280_720.tar.gz"

Running ISS Memory to memory WDR Example

CCS connection:

- Load the generated bsp_examples_m2mIssWdr_m4vpss_release.xem4 file in CCS
- Run the application
- Load any 1280x960 size Bayer RAW image with the format GRBG at the given location using CCS scripting console.
- Save the buffer with the command mentioned in the log and verify the buffer.

Build:

- Building for EVM, select **PLATFORM := tda3xx-evm**, **PACKAGE_SELECT := all** and **INCLUDE_WDR_LDC := yes** while building the binary

Note:

- **For using this example, install add on package for ISS on starterware. Details can be found in starterware releasenotes/userguide. After installing starterware ISS add on package, please make sure that LDC and WDR files vpshal_issglbce.c and vpshal_issldc.c are present in the folder starterware_\vpslib\hal\src**
- Sample Reference Input files for ISS M2M WDR Sample application is present at "bspdrivers_\docs\test_inputs\ISSM2MWdr_BayerRAW_GRBG_1280_960.tar.gz"

Running ISS Memory to memory ReSizer Example

This demo application demonstrates how to use multiple instances of ISP driver in parallel. In this demo application, an task uses ISP to convert from RAW 12 to YUV images and another task to re-size YUV420 SP images.

CCS connection:

- Load the generated bsp_examples_m2mIss_m4vpss_release.xem4 file in CCS
- Run the application
- Load any 1280x736 size Bayer RAW image with the format GRBG at the given location using CCS scripting console.
- Load any 1280x480 size YUV420 SP (NV12) image at the given location using CCS scripting console. This frame will be scaled upto 1920x1080
- Save the buffer with the command mentioned in the log and verify the buffer.

Build:

- Building for EVM, select **PLATFORM := tda3xx-evm**, **PACKAGE_SELECT := all** and **INCLUDE_WDR_LDC := yes** while building the binary

Note:

- **For using this example, install add on package for ISS on starterware. Details can be found in starterware releasenotes/userguide. After installing starterware ISS add on package, please make sure that LDC and WDR files vpshal_issglbce.c and vpshal_issldc.c are present in the folder starterware_\vpslib\hal\src**
- Sample Reference Input files for ISS M2M RSZ Sample application is present at "bspdrivers_\docs\test_inputs\IssM2MRszInput.tar.gz"

Running ISS Memory to memory SIMCOP Example

CCS connection:

- Load the generated bsp_examples_m2mSimcopLdcVtnf_m4vpss_release.xem4 file in CCS
- Run the application
- Load required mesh table and 4 images of resolution 1920X1080 of type YUV420 (NV12) using CCS scripting console.
  - The test images that we used, is available at bspdrivers_\docs\test_inputs\issSimcopTestImages.tar.gz
  - BowerFisheye.dat.bin is the Mesh Table
  - IMG_G1X_Bower_0001_420_mult.yuv is set of 4 images
  - LDC_VTNF_Output_YUV420_DRV.yuv is the expected result after processing

Build:

- Building for EVM, select **PLATFORM := tda3xx-evm**, **PACKAGE_SELECT := all** and **INCLUDE_WDR_LDC := yes** while building the binary

Note:

- **For using this example, install add on package for ISS on starterware. Details can be found in starterware releasenotes/userguide. After installing starterware ISS add on package, please make sure that LDC and WDR files vpshal_issglbce.c and vpshal_issldc.c are present in the folder starterware_\vpslib\hal\src**

Running BSP Application on TDA2XX EVM

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

Common Steps for all examples

- Installing CCS
  - CCSv5.4.0 Release Candidate 2 (Build 91) at http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v5
  - EMU_SR release M03 via update manager, containing latest debug drivers and trace software components. Installation instructions below:http://ap-fpdsp-swapps.dal.design.ti.com/index.php/Emulation_Debug_Software_Updates#Update_Installation_Steps
  - DRA7xx Chip Support Package: device xml files, GEL files, etc. for CCS configuration https://dncsps.itg.ti.com/sites/CS-IC-doc/ddt/Development%20and%20Debug%20Tools%20%20%20%20%20%20VAYUDRIAUTO/Forms/AllItems.aspx
- Create a new Target Configuration file with device as DRA7XX and specify the Emulator type which you are using.
- Once Tda2xx is launched with the target configuration created , it will automatically load all the Gel files required.
- Connect to A15 core 0 and bring benellie out of reset(M4_IPU1), run IPU1SSClkEnable_API from Gel scripts.
- Load and Run the respective examples
- In case of ADV7611 HDMI Receiver (> REV D boards) capture through VIP, SW8000[1:4] should be set to 0011. EDID programming to the EEPROM is not required as the ADV7611 driver uses the internal EDID RAM.
- For VIP capture from Multi-deserializer board, the multi-deserializer board should be be configured for 6-channel operation. The CN2, CN3 and CN4 jumper settings should be set as per below picture

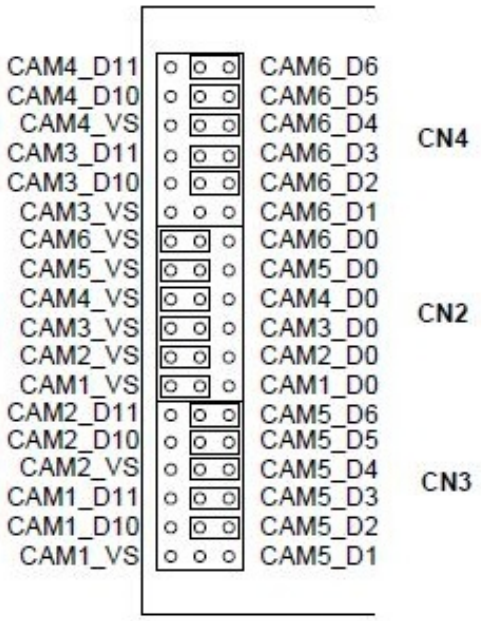


Figure 2.3: Configuration headers - “All 6 channels - 8b data” position

Running Capture example

- Make sure OV sensor OV1063x is connected to the vision app board before running the example.
- For all the options of capture example, Switch settings in Vision App board should set to SW3[1-8] - "xxxxxxx0". i.e. disable DIP switch override as this is now controlled through GPIO in board driver
- For options with HDMI Capture using SII, follow these steps

1. Switch settings in Vision App board should be set to SW1 - "1111",i.e enable all the dip switches in SW1. Now the programming is controlled through software
2. Run the edid_programmer binary from A15
3. Reset the switch settings to SW1 - "0000", i.e switch off all the dip switches, and run the application now

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option.
- Save the buffer with the command mentioned in the log and verify the buffer.

Constraints:

- For running on EVM, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-evm**

Running Display example

Configuring LCD Display:

- Make sure the LCD panel is connected to the base EVM before running the example binary.
- Make sure the HDMI is connected to TV from the EVM for HDMI display options.

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option and load the buffer as suggested.
- Content will be displayed on the LCD panel connected to EVM for options till 5
- Content will be displayed on the HDMI TV connected to EVM for options 6 and 7

Constraints:

- For running on EVM, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-evm**

Note:

- Test Input files for Display Sample application will be in "bspdrivers_\docs\test_inputs\DisplayInput.rar"
- For Option 1 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf"
- For Option 2 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf"
- For Option 3 : Load the input file "Logo_rgb888_prog_packed_1920_1080.tigf"
- For Option 4 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf" for YUV422I_YUYV frames and "Logo_rgb888_prog_packed_1920_1080.tigf" for BGR24_888 frames
- For Option 5 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf" for YUV422I_YUYV frames and "Logo_rgb888_prog_packed_1920_1080.tigf" for BGR24_888 frames
- For Option 6 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf"
- For Option 7 : Load the input file "PSP4_yuyv422_prog_packed_1920_1080.tigf"

Running M2MVpe example

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application
- Load the input file "VpeInputFlag10_nv12_prog_packed_720_240.tigf" as suggested.
- Save the buffer with the command mentioned in the log and verify the buffer.

Constraints:

- For running on EVM, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-evm**

Note:

- Test Input files for M2MVpe Sample application will be in "bspdrivers_\docs\test_inputs\VpeInputFlag10_nv12_prog_packed_720_240.rar"

Running Loopback example

- Make sure OV sensor is connected to vision daughter card and LCD Panel to the Base EVM before running the binary.
- For all the options of Loopback examples, Switch settings in Vision App board should set to SW3[1-8] - "xxxxxxx0". i.e. disable DIP switch override as this is now controlled through GPIO in board driver

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option.
- Captured Content will be displayed on the LCD panel connected to EVM

Constraints:

- For running on EVM, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-evm**

Running I2C LED Blink example

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option.
- LEDs on Vayu CPU Board will be blinking

Build:

- For running on EVM, select **PLATFORM := tda2xx-evm** while building the binary

Running UART example

- Make sure mini USB Cable is connected to host PC and the EVM, configure a serial terminal like Teraterm for standard baud rate(115200) and connect on newly installed Port.

CCS connection:

- Load the generated Xem4 file in CCS
- Run the application, Select any of the option.
- Input a text of 1k bytes for RX, you will see the text characters on terminal for TX

Build:

- For running on EVM, select **PLATFORM := tda2xx-evm** while building the binary

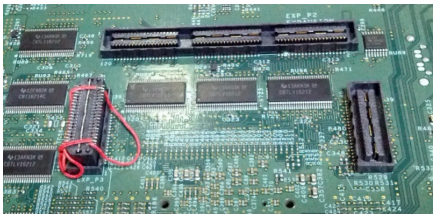
Running McSPI example

- For spi1tospi2 example Make sure the pins of SPI1 and SPI2 instances on the EVM are blue wired and board is configured for TX/RX Connections.

For TDA2xx and TDA2Ex please find the images below for connections. SPI pins to be connected to daughter card connector EXP_P3 present at bottom side of the EVM.

SPI1 to SPI2

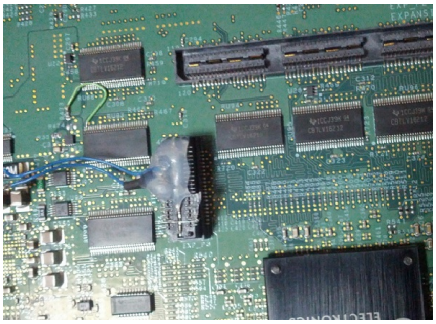
SPI1	SPI2
SPI1 CS0 (Pin 36)	SPI2 CS0 (Pin 2)
SPI1 SCLK (Pin 35)	SPI2 SCLK (Pin 1)
SPI1 MOSI (Pin 38)	SPI2 MISO (Pin 3)
SPI1 MISO (Pin 37)	SPI2 MOSI (Pin 4)



- For loopback example Make sure the Tx and Rx pins of the SPI instance to be tested are connected on the EVM and board is configured for TX/RX Connections. Loopback test is supported on SPI1 and SPI2 instances.

SPI LOOPBACK

SPI1 MOSI (Pin 38)	SPI1 MISO (Pin 37)
SPI2 MISO (Pin 3)	SPI2 MOSI (Pin 4)



CCS connection:

- Load the generated Xem4 file in CCS
- Run the application.
- Tx and Rx buffers are verified for data integrity, and the result is printed on console.

Build:

- For running on EVM, select **PLATFORM := tda2xx-evm** while building the binary

Running Audio Sinetone example

- Make Sure you have speakers or Headphones connected to the Headphone socket on board.

CCS connection:

- Connect A15 Core and get DSP Core out of reset by executing script in scripts->DRA7XX MULTICORE Initialization -> DSP1ClkEnable_API.
- Connect to DSP1 Core
- Run A15 Core (Known Issue)
- Load and Run the generated xe66 binary in CCS on DSP1 Core.
- You should hear sine tone

Build:

- Build Command is to be as here **gmake -s bsp_examples_audio PLATFROM=tda2xx-evm CORE=c66xdsp**

Running Audio Loopback example

- Make Sure you have speakers or Headphones connected to the Headphone socket on board.
- Connect Line In Cable between "PC/Audio Device and Line In Port of EVM"
- Start Playing Audio in the Audio Device

CCS connection:

- Connect A15 Core and get DSP Core out of reset by executing script in scripts->DRA7XX MULTICORE Initialization -> DSP1ClkEnable_API.
- Connect to DSP1 Core
- Run A15 Core (Known Issue)
- Load and Run the generated xe66 binary in CCS on DSP1 Core.
- You should hear Audio which you are playing in the audio device.

Build:

- Build Command is to be as here **gmake -s bsp_examples_audio_loopback PLATFROM=tda2xx-evm CORE=c66xdsp**

Running BSP Application on TDA2EX EVM

- The steps to run BSP examples for TDA2EX is similar to that of TDA2XX
- In case of VIP capture from various sources from VISION card, software mux control is not supported in TDA2EX. Hence the VIDEO CONFIG Switch (SW3) present in the VISION card should be used to select video source to VIP (VIN2A) as per below table

TDA2EX VISION Card Mux

TDA2EX VISION Source	SW3[1:8] Settings
HDMI input P5 (ADV7611/SIL9127)	OFF OFF ON ON OFF ON OFF ON
LI Imager P2	ON ON OFF ON OFF ON OFF ON
OV Imager P1	OFF ON OFF ON OFF ON OFF ON
Aptina Imager P3	ON OFF OFF ON OFF ON OFF ON
FPD In (Multi-serdes)	OFF OFF ON OFF OFF ON OFF ON

- In case of ADV7611 HDMI Receiver (> REV D boards) capture through VIP, SW8000[1:4] should be set to 0011. EDID programming to the EEPROM is not required as the ADV7611 driver uses the internal EDID RAM.

Changes needed for 4 Channel Multi-deserializer on TDA2EX EVM

Below are the board level changes required for using TDA2XX Vision board and Multi-deserializer board to capture 4 channel LVDS through VIP. All the board level muxing changes and pin mux configurations are already taken care in BSP board module. So they are not mentioned here.

Apart from the below board modifications, CPLD1 and CPLD2 present in the VISION card should be re-programmed with the CPLD for TDA2EX multi-deserializer present in `$(BSP_INSTALL_DIR)\docs\tda2ex\vision_cpld` folder

VIN1A (CAM2, connected to VIN5A pads):

- No changes required.
- The pin mapping is shown in below table

TDA2EX Multi-deserializer CAM2->VIN1A Pin Mapping

TDA2EX Pin	TDA2EX Function	Vision Net
GPIO6_10	VIN1A_CLK	VIN5A_CLK0
MMC3_D6	VIN1A_HSYNC	VIN5A_HSYNC
MMC3_D7	VIN1A_VSYNC	VIN5A_VSYNC
MMC3_D5	VIN1A_D0	VIN5A_D0
MMC3_D4	VIN1A_D1	VIN5A_D1
MMC3_D3	VIN1A_D2	VIN5A_D2
MMC3_D2	VIN1A_D3	VIN5A_D3
MMC3_D1	VIN1A_D4	VIN5A_D4
MMC3_D0	VIN1A_D5	VIN5A_D5
MMC3_CMD	VIN1A_D6	VIN5A_D6
MMC3_CLK	VIN1A_D7	VIN5A_D7

VIN1B (CAM4, connected to VIN4B pads):

- On TDA2EX EVM, install R-packs RN17, RN18 and RN19 with 0 ohm resistors
- On TDA2EX EVM, for GPMC_A8 pin: Remove R501 and add wire from R501 (away from mux) to EXP_P2 pin 62
- The pin mapping is shown in below table

TDA2EX Multi-deserializer CAM4->VIN1B Pin Mapping

TDA2EX Pin	TDA2EX Function	Vision Net
MDIO_MCLK	VIN1B_CLK	VIN4B_CLK
GPMC_A8	VIN1B_HSYNC	VIN2A_HSYNC
RGMII0_TXD1	VIN1B_VSYNC	VIN4B_VSYNC
MDIO_D	VIN1B_D0	VIN4B_D0
UART3_RXD	VIN1B_D1	VIN4B_D1
UART3_TXD	VIN1B_D2	VIN4B_D2
RGMII0_TXC	VIN1B_D3	VIN4B_D3

RGMII0_TXCTL	VIN1B_D4	VIN4B_D4
RGMII0_RXC	VIN1B_D5	VIN4B_D5
RGMII0_RXCTL	VIN1B_D6	VIN4B_D6
RGMII0_RXD3	VIN1B_D7	VIN4B_D7

VIN2A (CAM1, connected to VIN2A pads):

- On TDA2EX EVM, for RGMII0_TXD2 pin: Remove R506 and add wire from R506 (away from mux) to EXP_P2 pin 75
- The pin mapping is shown in below table

TDA2EX Multi-deserializer CAM1->VIN2A Pin Mapping

TDA2EX Pin	TDA2EX Function	Vision Net
VIN2A_CLK	VIN2A_CLK	VIN1A_CLK
RGMII0_TXD2	VIN2A_HSYNC	VIN1B_CLK1
VIN2A_VSYNC	VIN2A_VSYNC	VIN1A_VSYNC
VIN2A_D[7:0]	VIN2A_D[7:0]	VIN1A_D[7:0]

VIN2B (CAM3, connected to VIN2B pads):

- On TDA2EX EVM, for GPIO6_11 pin: Remove R638, R647, R786 and install 0-ohm at R1099
- On TDA2EX EVM, for VIN2A_FLD pin: Remove RU40 and wire around - connect 2 to 4
- On VISION Card, for VIN2A_FLD pin: Add wire from TP8 to TP48
- The pin mapping is shown in below table

TDA2EX Multi-deserializer CAM3->VIN2B Pin Mapping

TDA2EX Pin	TDA2EX Function	Vision Net
VIN2A_FLD0	VIN2B_CLK	TEST POINT (TP8)
VIN2A_HSYNC	VIN2B_HSYNC	VIN1A_HSYNC
GPIO6_11	VIN2B_VSYNC	VIN3A_HSYNC_VIN2A_D22
VIN2A_D[16:23]	VIN2B_D[7:0]	VIN1A[23:16] (Note: bit order is reversed)

Running BSP Application on TDA2XX Virtio

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

Common Steps for all examples

- Install Innovator and the respective Virtio package as per the steps given in the Virtio release/user guide
- Open the CCS and connect to TDA2xx (IPU1_Core0) using the CCS and debugger.
- Load the GEL file present at `$BSP_INSTALL_DIR/docs/tda2xx/TDA2xx_virtio_config.gel`. Run option `M4_MMuConfig` from the menu.

This will configure the IPU MMU.

- Load and Run the respective examples

Running Capture example

- Edit the VIP superfile present in `$CCS_Install/ccs_base_5.x.x/simulation_csp_vayu/vip_input_superfile_vayu.txt` like below pointing to the right .bin file containing BT656 image.

```
1   D:\Images\sim_capture_input_files\output_bt656_QCIF.bin
2   D:\Images\sim_capture_input_files\output_bt1120_QCIF.bin
3
D:\Images\sim_capture_input_files\vayu_output_bt1120_QCIF_RGB_repack_0.bin
4
D:\Images\sim_capture_input_files\vayu_output_bt1120_QCIF_YUV444_repack_0.bin
```

- Find and replace the path for the option `SUPERFILE_NAME` to the above super file path in file `tisim_init.cfg` present in `C:\Virtio\Platforms\VPVayu5.3`
- Run the capture application and it will capture 128x128 frame size from the BT656 bin file

Running Display example

Configuring LCD Display:

- In virtio dumb-panel resolution is by default 240 * 320, it can the changed as follows
- Browse to **VPVayu::SDP::Vayu::DumbPanel**.
- Right-click Show Parameters and you will find LCD_Height and LCD_Width parameters.Set Resolution to 720 * 480.
- In Virtio Frame change is very fast because DSS interrupts are too fast in comparison with M4 speed on virtio, thus resulting in task starvation. So Display Controller refresh rate needs to be changed to higher value. Steps as follows:
- Browse to **VPVayu::SDP::Vayu::Displaycontroller**
- Change the **Refresh rate** to **200000** instead of current 20000

Data format - RGB 24 bit. Width and Height - Auto.

CCS connection:

- start the Vayu Target and connect to virtio
- Load the generated Xem4 file in CCS
- Run the application, Select any of the option and load the buffer as suggested.
- Content will be displayed on Dumb Panel* in virtio

Constraints:

- For running on Virtio, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-virtio**
- Blending in Virtio is currently not supported, Thus option 2 and 3 with GFX pipeline where color key and global alpha is enabled displays background color for the color key instead of showing the layer below.

Running Loopback example

Configuring LCD Display:

- Follow steps from **Running Capture example** and **Running Display example**

Constraints:

- For running on Virtio, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-virtio**
- Select option 0, Only virtio option will run.

Running BSP Application on TI814x

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

Demo application on Video Surveillance (VS) board

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

- Single channel NTSC interlaced input to VIP capture driver using TVP5158.
- VIP capture outputs all the buffers in the memory in YUV422 interleaved format.

Steps to run the Demo on VS 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_2x_evm_A8_ddr3.gel*** under directory *$BSP_INSTALL_DIR/docs/ti814x* for A8 processor.
- Run *Scripts* > *TI814x HDVPSS Init* > *HDVPSSInit*. This will enable the DDR, Ducati and HDMI.
- Connect to CortexM3_ISS

Steps for specific application

- Connect 1 input source to the composite input of the VS application board.
- Load and Run $BSP_INSTALL_DIR/bspdrivers_/build/bsp_examples_captureVip/bin/ti814x-evm/bsp_examples_captureVip_m3vpss_debug.x
- Select option 5 at " Enter Test to Run:" option.

Demo application on Vision Application board

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

- Single channel YUV422 capture from the sensor MT9V022.
- VIP capture outputs all the buffers in the memory in YUV422 interleaved format.

Steps to run the Demo on Vision Application 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_2x_evm_A8_ddr3.gel*** under directory *$BSP_INSTALL_DIR/docs/ti814x* for A8 processor.
- Run *Scripts* > *TI814x HDVPSS Init* > *HDVPSSInit*. This will enable the DDR, Ducati and HDMI.
- Connect to CortexM3_ISS

Steps for specific application

- Connect the aptina sensor MT9V022 to the JP1/JP2 connector on the Vision Application Board
- Configure the S1 DIP switch to route the Aptina sensor data to the VIP. The table to configure the switch is as given below.

- Load and Run $BSP_INSTALL_DIR/bspdrivers_/build/bsp_examples_captureVip/bin/ti814x-evm/bsp_examples_captureVip_m3vpss_debug.xe
- Select option 4 at " Enter Test to Run:" option.

Switch settings for VIP 0 port A

Device	DIP_SW2	DIP_SW1	DIP_SW0
Omni Vision OV10630	0	0	0
Aptina connector JP2	0	0	1
Aptina connector JP1	0	1	0
TVP 5158 (VIN1)	0	1	1
TVP 5158 (VIN2)	1	0	0
TVP 5158 (VIN3)	1	0	1
TVP 5158 (VIN4)	1	1	0

Switch settings for VIP1 port A (Using a jumper the pins 17 & 19 are shorted on the JP13 connector)

Device	DIP_SW8 (Pin 19 of JP13)	DIP_SW7	DIP_SW6
Omni Vision OV10630	0	0	0
Aptina connector JP2	0	0	1
Aptina connector JP1	0	1	0
TVP 5158 (VIN1)	0	1	1
TVP 5158 (VIN2)	1	0	0
TVP 5158 (VIN3)	1	0	1
TVP 5158 (VIN4)	1	1	0

Running BSP Application on Zebu

- The BSP applications are validated on Zebu release as mentioned in the release notes
- The executable for M4 needs to be renamed to have an .elf extension
- In addition to the M4 executable, an A15 executable is also required to be run. This enables PRCM for IPU, VIP, DSS, VPE as well as other system peripherals. It also sets up the IPU MMU with following settings. This is present in $BSP_INSTALL_DIR/docs/tda2xx/prcm_mmu_CxA15.elf

```
   #define BEN1_BASE_ADDR              0x58800000
#define BEN1_MMU_CFG                (BEN1_BASE_ADDR + 0x00080000)
```

```
   /*Large Page Translations */
/* Logical Address */
WR_MEM_32(BEN1_MMU_CFG+0x800,         0x8F000000);
WR_MEM_32(BEN1_MMU_CFG+0x804,         0x80000000);
WR_MEM_32(BEN1_MMU_CFG+0x808,         0xC0000000);
WR_MEM_32(BEN1_MMU_CFG+0x80C,         0x60000000);
```

```
   /* Physical Address */
WR_MEM_32(BEN1_MMU_CFG+0x820,         0x8F000000);
WR_MEM_32(BEN1_MMU_CFG+0x824,         0x80000000);
```

```
WR_MEM_32(BEN1_MMU_CFG+0x828,          0xC0000000);
WR_MEM_32(BEN1_MMU_CFG+0x82C,          0x60000000);
```

```
   /* Policy Register */
WR_MEM_32(BEN1_MMU_CFG+0x840,          0x00000007);
WR_MEM_32(BEN1_MMU_CFG+0x844,          0x000B0007);
WR_MEM_32(BEN1_MMU_CFG+0x848,          0x00020007);
WR_MEM_32(BEN1_MMU_CFG+0x84C,          0x00020007);
```

```
   /*Medium Page*/
/* Logical Address */
WR_MEM_32(BEN1_MMU_CFG+0x860,          0x00300000);
WR_MEM_32(BEN1_MMU_CFG+0x864,          0x00400000);
```

```
   /* Physical Address */
WR_MEM_32(BEN1_MMU_CFG+0x8A0,          0x40300000);
WR_MEM_32(BEN1_MMU_CFG+0x8A4,          0x40400000);
```

```
   /* Policy Register */
WR_MEM_32(BEN1_MMU_CFG+0x8E0,          0x00000007);
WR_MEM_32(BEN1_MMU_CFG+0x8E4,          0x00020007);
```

```
   /*Small Page*/
/* Logical Address */
WR_MEM_32(BEN1_MMU_CFG+0x920,          0x00000000);
WR_MEM_32(BEN1_MMU_CFG+0x924,          0x40000000);
WR_MEM_32(BEN1_MMU_CFG+0x928,          0x00004000);
WR_MEM_32(BEN1_MMU_CFG+0x92C,          0x00008000);
WR_MEM_32(BEN1_MMU_CFG+0x930,          0x20000000);
```

```
   /* Physical Address */
WR_MEM_32(BEN1_MMU_CFG+0x9A0,          0x55020000);
WR_MEM_32(BEN1_MMU_CFG+0x9A4,          0x55080000);
WR_MEM_32(BEN1_MMU_CFG+0x9A8,          0x55024000);
WR_MEM_32(BEN1_MMU_CFG+0x9AC,          0x55028000);
WR_MEM_32(BEN1_MMU_CFG+0x9B0,          0x55020000);
```

```
   /* Policy Register */
WR_MEM_32(BEN1_MMU_CFG+0xA20,          0x0001000B);
WR_MEM_32(BEN1_MMU_CFG+0xA24,          0x0000000B);
WR_MEM_32(BEN1_MMU_CFG+0xA28,          0x00010007);
WR_MEM_32(BEN1_MMU_CFG+0xA2C,          0x00000007);
WR_MEM_32(BEN1_MMU_CFG+0xA30,          0x00000007);
```

Zebu command to use:

bin/runZeBu prcm_mmu_CxA15.elf -CM4_IPU1 <example>.elf

Viewing application printfs:

In the VMP control panel menu, goto **Components -> Uart1_Terminal**. In the UART terminal menu, goto **Uart -> Baud Rate -> 3686400 Bds**

Running Capture example

Zebu command to use:
bin/runZeBu prcm_mmu_CxA15.elf -CM4_IPU1 bsp_examples_captureVip_m4vpss_debug.elf

Enabling camera:
In the VMP control panel menu, goto **Components** -> **Camera_CCP1** In the Camera control panel, goto **File** -> **Test Pattern** -> **B & W Fade** In the Camera control panel, goto **File** -> **Enable** to start the camera.

Constraints:

- For running on Zebu, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-zebu**
- In **bspdrivers_\examples\utility\bspCommonBIOS.cfg**, a special mapping is used to enable loading executables without CCS.
- Only *Option 8*' of VIP capture application is supportted on Zebu
- In the Camera control panel, click on the **Settings Icon** to select width and height on the input signal. This should be configured to be less than 1920x1080. Preferred size is 640x480 or 720x480.
- Note that it takes around 10-15 mins to reach the application menu on UART console and another 15-20 mins for the application to complete

Running Display example

Zebu command to use:
bin/runZeBu prcm_mmu_CxA15.elf -CM4_IPU1 bsp_examples_displayDss_m4vpss_debug.elf

Enabling LCD Display:

- In the VMP control panel menu, goto **Components** -> **Screen LCD**.
- In the Screen LCD control panel, click on power symbol to turn on LCD screen.
- In the Screen LCD control panel, goto **Screen** -> **Custom** to choose Custom LCD panel.
- In the Screen LCD control panel, click on **setting Icon** and set the following parameters.

Data format - RGB 24 bit.
HSYNC Polarity - Normal.
VSYNC Polarity - Normal.
Width and Height - Auto.

Loading Frame Buffers in memory

- Copy the TCL script(mem_load_zebu.tcl) and C executable(DDRsplit) from $BSP_INSTALL_DIR/docs/tda2xx/ to a folder.
- Do a “chmod 777” for the C executables.
- When the Zebu model is launched before sourcing and executing the TCL script rename the Zebu DDR memories (banked) to the names by which they are referred in the TCL script. (This is a one time configuration and will be maintained when launching multiple times.) For renaming open the memory window in VMP and search for DDR3. Rename the last 8 memories to

Vayu_0_even
Vayu_0_odd
Vayu_1_even
Vayu_1_odd
Vayu_2_even
Vayu_2_odd
Vayu_3_even
Vayu_3_odd

according to the memory hierarchy name.

- Source the TCL file “source mem_load_zebu.tcl” in the VMP command prompt.
- When Application requests to load the buffers execute the command “memory_write <filename> 0x<startAddr in Hex>” in the VMP command promt.

Constraints:

- For running on Zebu, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-zebu**
- In **bspdrivers_\examples\utility\bspCommonBIOS.cfg**, a special mapping is used to enable loading executables without CCS.
- Only *Option 8*' of VIP capture application is supportted on Zebu
- Note that it takes around 10-15 mins to reach the application menu on UART console and another 15-20 mins for the application to complete

Running Loop Back example

Zebu command to use:
bin/runZeBu prcm_mmu_CxA15.elf -CM4_IPU1 bsp_examples_loopback_m4vpss_release.elf

Enabling camera:
In the VMP control panel menu, goto **Components** -> **Camera_CCP1** In the Camera control panel, goto **File** -> **Test Pattern** -> **B & W Fade** In the Camera control panel, goto **File** -> **Enable** to start the camera.

Enabling LCD Display:

- In the VMP control panel menu, goto **Components** -> **Screen LCD**.
- In the Screen LCD control panel, click on power symbol to turn on LCD screen.
- In the Screen LCD control panel, goto **Screen** -> **Custom** to choose Custom LCD panel.
- In the Screen LCD control panel, click on **setting Icon** and set the following parameters.

Data format - RGB 24 bit.
HSYNC Polarity - Normal.
VSYNC Polarity - Normal.
Width and Height - Auto.

Constraints:

- For running on Zebu, in Rules.make, ensure **PACKAGE_SELECT := all** and **PLATFORM := tda2xx-zebu**
- In **bspdrivers_\examples\utility\bspCommonBIOS.cfg**, a special mapping is used to enable loading executables without CCS.
- Only *Option 1*' of Loop Back application is supportted on Zebu
- In the Camera control panel, click on the **Settings Icon** to select width and height on the input signal. Preferred size is 720x480.
- Note that it takes around 10-15 mins to reach the application menu on UART console and another 35-40 mins for the application to complete

Compiling BSP Drivers

BSP drivers and examples can be built using the *Makefile* present in the BSP installation directory. Before doing so, user may have to modify the *Rules.make* file (present in the installation directory), depending upon the 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_M4** := Points to the Codegen toolchain for M4.
- **CODEGEN_PATH_M3** := Points to the Codegen toolchain for M3.

- **CODEGEN_PATH_DSP**:= Points to the Codegen toolchain for DSP.
- **bsp_PATH** := Points to the BSP installation directory.
- **bios_PATH** := Points to the BIOS installation directory.
- **xdc_PATH** := Points to the XDC installation directory.
- **starterware_PATH** := Points to the starterware installation directory.
- **edma3_lld_PATH** := Points to the EDMA 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.

Provide the command `gmake -s all`. This will clean and recursively build all the libraries and examples for the default platform (tda2xx-evm) and default profile (release).

```
>gmake -s all
```

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 for convenience:

- **gmake -s bsp** - incrementally builds only BSP drivers
- **gmake -s examples** - incrementally builds BSP drivers and all audio, video and serial examples
- **gmake -s audio PLATFORM:=tda2xx-evm CORE=c66xdsp** - incrementally builds BSP drivers and all audio examples for TDA2XX
- **gmake -s audio PLATFORM:=ti814x-evm CORE=c6xdsp** - incrementally builds BSP drivers and all audio examples for TI814X
- **gmake -s clean** - clean all drivers and examples
- **gmake -s examplesclean** - clean all examples ONLY
- **gmake -s *example_name*** - incrementally builds BSP drivers and the specific example ONLY. Values for *example_name* can be - captureVip, m2mVpe etc.
- **gmake -s iss** - incrementally builds BSP drivers and the ISS specific example ONLY. **Ensure to include INCLUDE_WDR_LDC := yes, if WDR & LDC examples is required.**
  - **If WDR and or LDC examples is required, ensure add on package for ISS on starterware is installed. Details can be found in starterware releasenotes/userguide.**

Modular build Configuration:

- **gmake -s example_name PACKAGE_SELECT=package_name** - incrementally builds the specific drivers and example ONLY. Values for *example_name* can be - captureVip, m2mVpe etc.Values for *package_name* can be - vps-vip-only,vps-vpe-only etc.
  - vps-vip-only - incrementally builds the vip driver library ONLY
  - vps-vpe-only - incrementally builds the vpe driver library ONLY
  - vps-dss-only - incrementally builds the dss driver library ONLY
  - vps-vip-dss - incrementally builds the vip and dss drivers library ONLY
  - vps-vip-vpe - incrementally builds the vip and vpe drivers library ONLY
  - vps-iss-only - incrementally builds the iss drivers library ONLY
  - vps-iss-dss-only - incrementally builds the iss and dssdrivers library ONLY

- all - incrementally builds all the libraries

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

Important
If the user is not building the drivers using the provided make system, the following macros should be defined at compile time. `-D___DSPBIOS___` `-D___DUCATI_FW___` `-DTDA3XX_BUILD` `-DTDA3XX_FAMILY_BUILD` `-DTDA2XX_BUILD` `-DTDA2XX_FAMILY_BUILD` `-DPLATFORM_EVM_SI` `-DUSE_STD_ASSERT` `-DTRACE_ENABLE` `-DVPS_VIP_BUILD` `-DVPS_VPE_BUILD` `-DVPS_DSS_BUILD` `-DRELEASE_BUILD` `-DASSERT_ENABLE` `-DBUILD_M4`
For compile and linker options used, refer the feature and performance datasheet.

Selective VIP build:

In case of SOC like Tda2xx where multiple VIP instances are present, the application could selective build any VIP instance by using below option while building BSP drivers

gmake -s $target PACKAGE_VIP1_BUILD="yes/no" PACKAGE_VIP2_BUILD="yes/no" PACKAGE_VIP3_BUILD="yes/no"

Example: To build only VIP1 use below command
gmake -s all PACKAGE_VIP1_BUILD=yes PACKAGE_VIP2_BUILD=no PACKAGE_VIP3_BUILD=no

Memory Map for TDA3XX

The following shows various sections defined and used by BSP drivers and its sample applications. This assumes a 512 MB of DDR memory. The Video M4 MMU is configured such away that the 512 MB DDR is split into two sections - 1st 512MB is cached and the next 512MB is non-cached (mapped to above 512MB physical).

- 1st 512MB - Cached:
  - VPSS M4 Data: IPU Core 0 bss and other data section (driver bss, const and other global variables reside here)
  - VPSS M4 Code: IPU Core 0 code program section (driver text section resides here)
  - Video M4 Data: IPU Core 1 bss and other data section (driver bss, const and other global variables reside here)
  - Video M4 Code: IPU Core 1 code program section (driver text section resides here)
  - Mirrored Section : This section is mapped to below mentioned 512MB
  - Frame Buffer: Frame buffer heap used for non-tiled video buffers
- 2nd 512MB - Non-Cached:
  - Mirrored Section : This section is mapped to above mentioned 512MB
  - VPDMA Desc Mem: BSP driver VPDMA descriptor memory section used to store descriptors and overlay memory

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

+-----------------+
| VPSS M4 Code    | (1MB - SBL_SIZE)
+-----------------+
| VPSS M4 Data    | 8MB
+-----------------+
| Video M4 Code   | 1MB
+-----------------+
| Video M4 Data   | 4MB
```

```
+-----------------+
| Mirrored Section| 2MB VPDMA Desc Mem
+-----------------+
|                 |
| Frame Buffer    | 240MB (Incase of 12x12, 48MB)
+-----------------+

DDR: 0xA0000000 (2nd 512MB - Non-Cached - but mapped to above 512MB
physical)

+-----------------+
| Mirrored Section| 14MB for Code + Data
+-----------------+
| VPDMA Desc Mem  | 2MB
+-----------------+
| Mirrored Section| 240MB Frame Buffer (Incase of 12x12, 48MB)
+-----------------+
```

Memory Map for TDA2XX

The following shows various sections defined and used by BSP drivers and its sample applications. This assumes a 1 GB of DDR memory. The Video M4 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:
  - VPSS M4 Data: VPSS M3 bss and other data section (driver bss, const and other global variables reside here)
  - VPSS M4 Code: VPSS M3 code program section (driver text section resides here)
- 2nd 512MB - Non-Cached:
  - Frame Buffer: Frame buffer heap used for non-tiled video buffers
  - Tiler 8-bit/16-bit: Tiler memory for the different tiler view
  - VPDMA Desc Mem: BSP driver VPDMA descriptor memory section used to store descriptors and overlay memory

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

+-----------------+
| Video M4 Code   | 2MB
+-----------------+
| Video M4 Data   | 10MB
+-----------------+
| NOT USED        | 500MB
+-----------------+


DDR: 0xA0000000 (2nd 512MB - Non-Cached)

+-----------------+
|                 |
| Frame Buffer    | 256MB
```

```
+-----------------+
| Tiler Buffer    | 128MB
+-----------------+
| VPDMA Desc Mem  | 2MB
+-----------------+
|     NOT USED    | 126MB
+-----------------+
```

Memory Map for TI814x

The following shows various sections defined and used by BSP 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 VPSS
  - 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: BSP driver VPDMA descriptor memory section used to store descriptors and overlay memory

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

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



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

Using BSP Utils

This section describes about BSP Utils . It shows how to run the BSP Utils. Currently, there are no Utils provided.

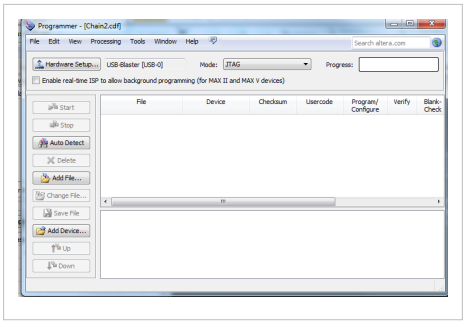
Directory Organization

The following expands on the directory structure of → BSP Folders

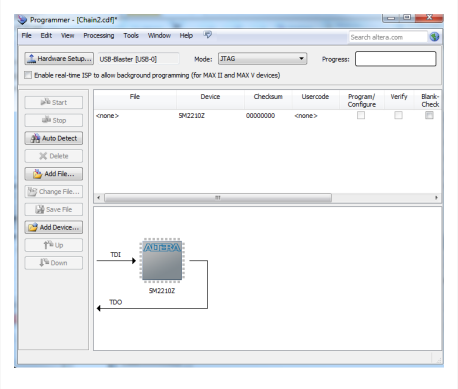
Programming CPLD for Tda2xx Vision Daughter Card

Steps for programming new file:

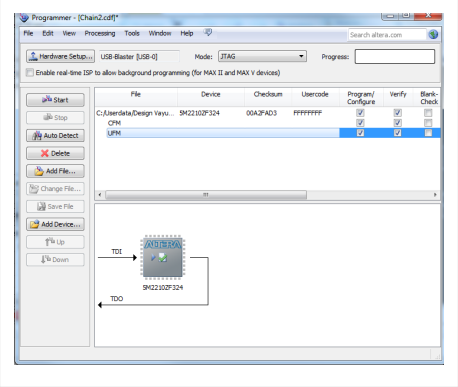
- Connect Altera USB Blaster cable to J2 header (for ‘CPLD1’) or J1 header (for 'CPLD2'). Make sure pin 1 on header matches pin 1 on cable (pod should extend away from board, not overlap).
- Apply power to system
- Launch Quartus II software.
- Under menu ‘Tools’ select Programmer.
- Make sure USB-Blaster is selected for Hardware (Shown below)



- Select ‘Auto Detect’ and the software will scan board looking for CPLD. It should find following:



- In top window, right click on top of ‘5M2210Z’ and select ‘Change File’. Browse to *.pof file provided in docs/tda2xx/vision_cpld folder.
- Make sure Program/Configure and Verify boxes are checked (shown below)…and then hit start button.



- Bar at top will show progress and will pass/fail when programming is complete.

Programming CPLD for Tda3xx EVM

Steps for programming the CPLD for Tda3xx is similar to Tda2xx Vision board as explained in previous section. Below are the changes to be taken care of

- Connect Altera USB Blaster cable to J1 header for programming both ‘CPLD1’ and/or ‘CPLD2’. Make sure pin 1 on header matches pin 1 on cable (pod should extend away from board, not overlap).
- Both the CPLD files are present in docs/tda3xx/cpld folder. Use REVB1 version.
- Select EPM2210 device and program the corresponding CPLD image
- Select EPM570 device and program the corresponding CPLD image

BSP Overview

This section provides top level information about BSP software architecture and the hardware architecture of VIP, VPE and DSS for TDA2XX and VIP and DSS for TDA3XX.

Hardware Overview

The VIP, VPE and DSS hardware overview explains the different hardware blocks in brief. It is not necessary to have a full knowledge of the hardware architecture to use BSP drivers. The hardware overview for TDA2XX can be found at TDA2XX Hardware Overview. The hardware overview for TDA3XX can be found at TDA3XX Hardware Overview.

BSP Software Overview

BSP software overview explains the BSP software and the major class of drivers supported by the BSP software interfaces to the application. BSP Software overview could be found at BSP 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

Platform & Board APIs and Drivers

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

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 BSP package can be found at → BSP Capture Driver UserGuide

Memory Driver

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 BSP package can be found at → BSP M2M VPE Driver UserGuide

Display Driver

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 BSP packages can be found at → BSP Display Driver UserGuide

Serial Drivers Examples Static Configurations

This section provides information about static configuration of Serial Drivers Examples, presently this is demonstrated in the cfg files of specific examples, but in case a user wants to use these for making them into a dynamic configuration or add to CCS project, they can refer to below section. In case user application has additional configurations, they need to take below as a reference and build on top of this, the configurations shown below are specifically for the applications which are present in the BSP Package. Below configurations would be easily understood if you have an understanding of BIOS IO Model

I2C Driver Application

Static Configuration needed for I2C Driver application is as below. This needs to be added in the application.

```
/* Use Semaphore, and Task modules and set global properties */
var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
Program.global.hsi2c_sem0 = Semaphore.create(1);
Program.global.hsi2c_sem1 = Semaphore.create(1);
Program.global.hsi2c_sem2 = Semaphore.create(1);
Program.global.hsi2c_sem3 = Semaphore.create(1);

var iomFxns = "I2c_IOMFXNS";
var initFxn = "user_i2c_init";
var deviceParams = "devInitParams";
var deviceId = 0;

GIO.addDeviceMeta("i2c", iomFxns, initFxn, deviceId, deviceParams);

var task0Params = new Task.Params();
task0Params.instance.name = "task0";
task0Params.stackSize = 0x2000;
Program.global.task0 = Task.create("&echo", task0Params);

/* configure the required Heap parameters */
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');
var prms = new HeapMem.Params;
prms.size = 8192;
var heap = HeapMem.create(prms);

var Memory = xdc.useModule('xdc.runtime.Memory');
Memory.defaultHeapInstance = heap;
```

Above is also demonstrated in examples\i2c\i2cSample.cfg

Audio Applications

Static configurations required for audio applications is shown as below, this is also mentioned in examples\audio\<example_name>\example_nameVayu.cfg

```
/* Use Semaphore, and Task modules and set global properties */
var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
Program.global.hsi2c_sem0 = Semaphore.create(1);
Program.global.hsi2c_sem1 = Semaphore.create(1);
Program.global.hsi2c_sem2 = Semaphore.create(1);
Program.global.hsi2c_sem3 = Semaphore.create(1);
```

```
var iomFxns = "I2c_IOMFXNS";
var initFxn = "user_i2c_init";
var deviceParams = "devInitParams";
var deviceId = 0;
GIO.addDeviceMeta("i2c", iomFxns, initFxn, deviceId, deviceParams);

var iomFxns = "I2c_IOMFXNS";
var initFxn = "user_i2c2_init";
var deviceParams = "devInitParams2";
var deviceId = 1;
GIO.addDeviceMeta("i2c", iomFxns, initFxn, deviceId, deviceParams);

var iomFxns = "Aic31_IOMFXNS";
var initFxn = "audioUserAic31Init";
var deviceParams = "audioAic31Params";
var deviceId = 0;
GIO.addDeviceMeta("/aic310", iomFxns, initFxn, deviceId, deviceParams);

var iomFxns = "Audio_IOMFXNS";
var initFxn = "audioUserAudioInit";
var deviceParams = "audioParams";
var deviceId = 0;
GIO.addDeviceMeta("/audio0", iomFxns, initFxn, deviceId, deviceParams);

var iomFxns = "Mcasp_IOMFXNS";
var initFxn = "audioUserMcaspInit";
var deviceParams = "audioMcaspParams";
var deviceId = 2;
GIO.addDeviceMeta("/mcasp2", iomFxns, initFxn, deviceId, deviceParams);

var task0Params = new Task.Params();
task0Params.priority = 5;
task0Params.instance.name = "task0";
Program.global.task0 = Task.create("&Audio_echo_Task", task0Params);

ECM.eventGroupHwiNum[0] = 7;
ECM.eventGroupHwiNum[1] = 8;
ECM.eventGroupHwiNum[2] = 9;
ECM.eventGroupHwiNum[3] = 10;

Program.sectMap[".text"] = "EXT_RAM";
Program.sectMap[".const"] = "EXT_RAM";
Program.sectMap[".plt"] = "EXT_RAM";
```

McSPI/UART Applications

McSPI/UART applications are already ported to use common bspCommonUtilityBIOS cfg file and additional configurations needed are given the respective application cfg files.

For example, if we take UART Application, the cfg file for this application content is as below

```
xdc.loadCapsule("utility/bspCommonBIOS.cfg");
var GIO = xdc.useModule('ti.sysbios.io.GIO');
var Task = xdc.useModule('ti.sysbios.knl.Task');
var IntXbar = xdc.useModule('ti.sysbios.family.shared.vayu.IntXbar');

var iomFxns = "Uart_IOMFXNS";
var initFxn = "UartApp_userInit";
var deviceParams = "uartParams";
var deviceId = 0;

GIO.addDeviceMeta("/uart0", iomFxns, initFxn, deviceId, deviceParams);

var task0Params = new Task.Params();
task0Params.instance.name = "task0";
task0Params.affinity = 0; /* Force to run on Core 0 */

Program.global.task0 = Task.create("&UartApp_sampleTask", task0Params);
```

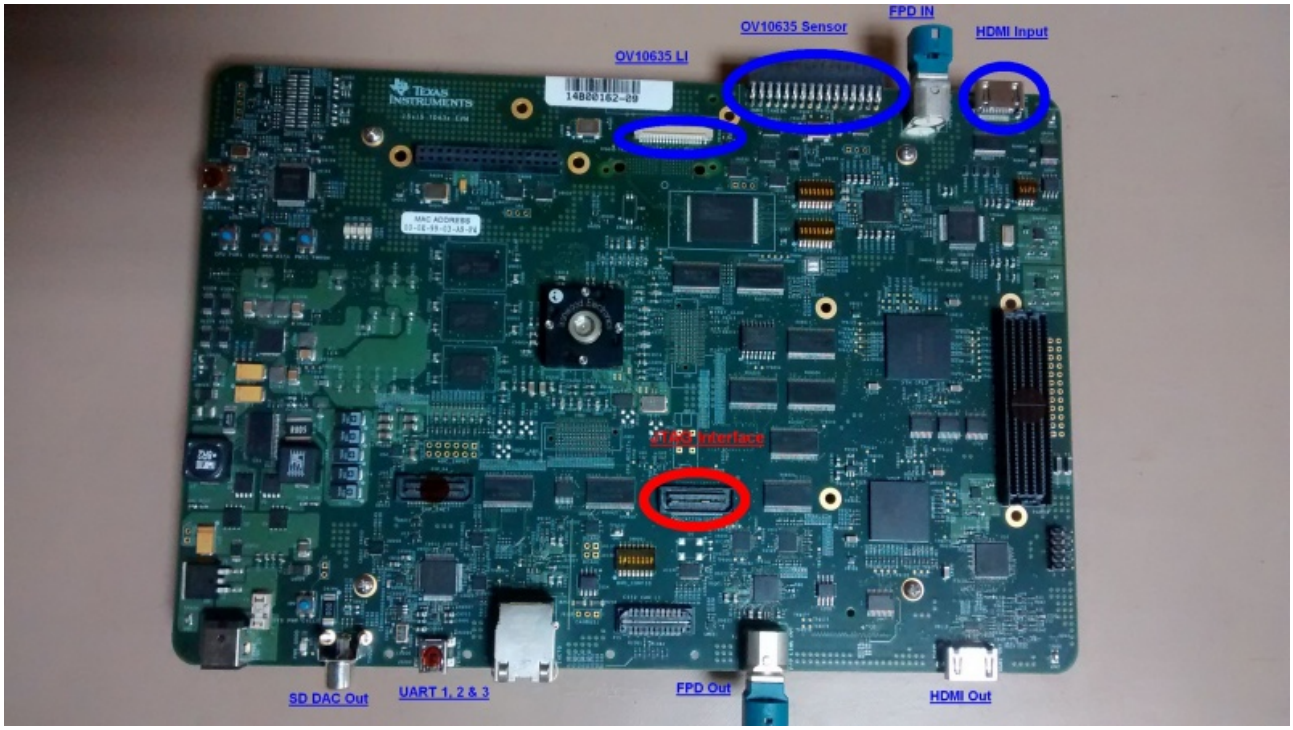
As mentioned above, common configuration is taken from bspCommonBIOS.cfg and all of remaining configuration needs to be added. Same is the case with McSPI as mentioned in examples\mcspi\mcspisample.cfg; examples\mcspiMasterSlave\<examplename>\mcspisample.cfg.

Connecting Sensors To EVM

TDA3xx

Interfaces

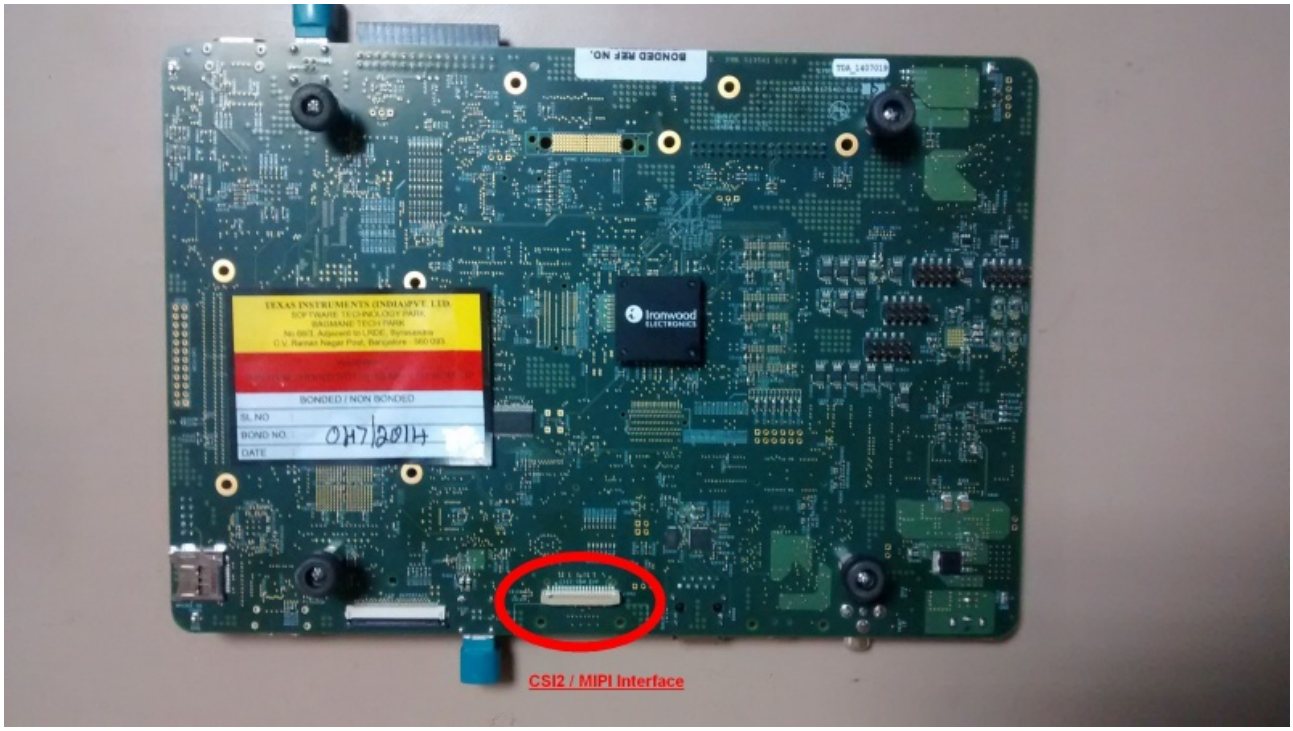
Display, Capture, JTAG interface, UART & FPD interfaces are highlighted below.



ISS MIPI Interface

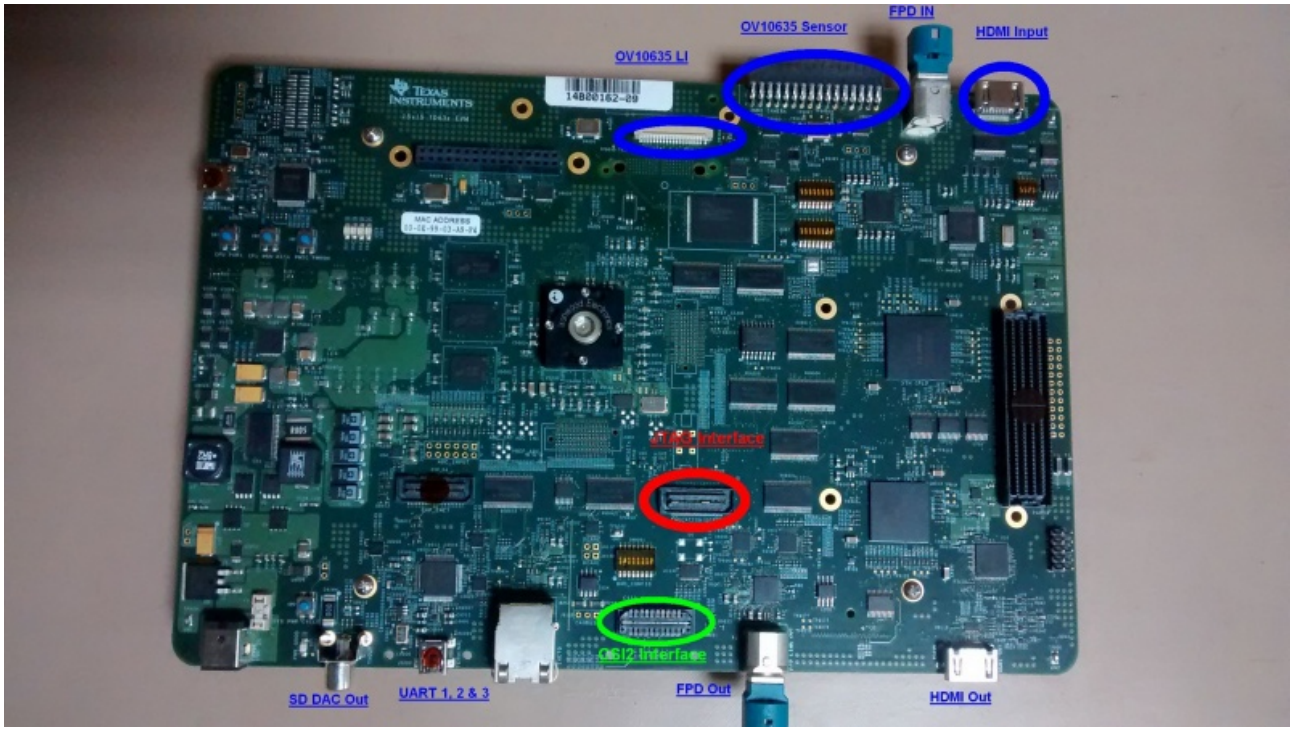
The CSI2 interface of the sensor should be connected to this header.

WARNING : Connecting NON MIPI Sensor to this interface, might potentially render the sensor un-usable.



ISS Parallel Interface

WARNING : Connecting MIPI sensor to Parallel Li connector, might potentially render the sensor un-usable.



MIPI Interface

The connector highlighted is green show high speed connector. This connects to TA3xx CSI2 interface and also LI Connector show in 'ISS MIPI Interface'

The UB960 EVM supported by BSP Drivers should be connected to this connector.

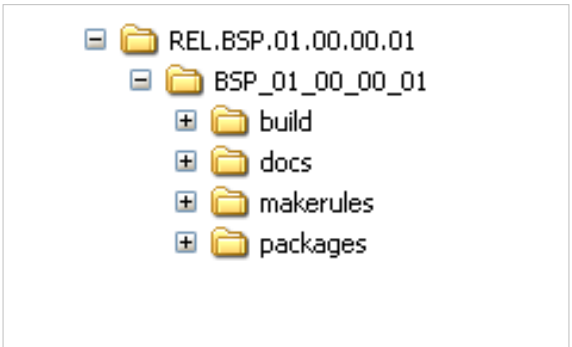
WARNING : An MIPI compliant sensor / source should be connected to either one of these interfaces. Connecting 2 different sensors / sources to these two connector is not permitted.

UserGuideBSPFolderOrg

BSP Code / Directory Organization

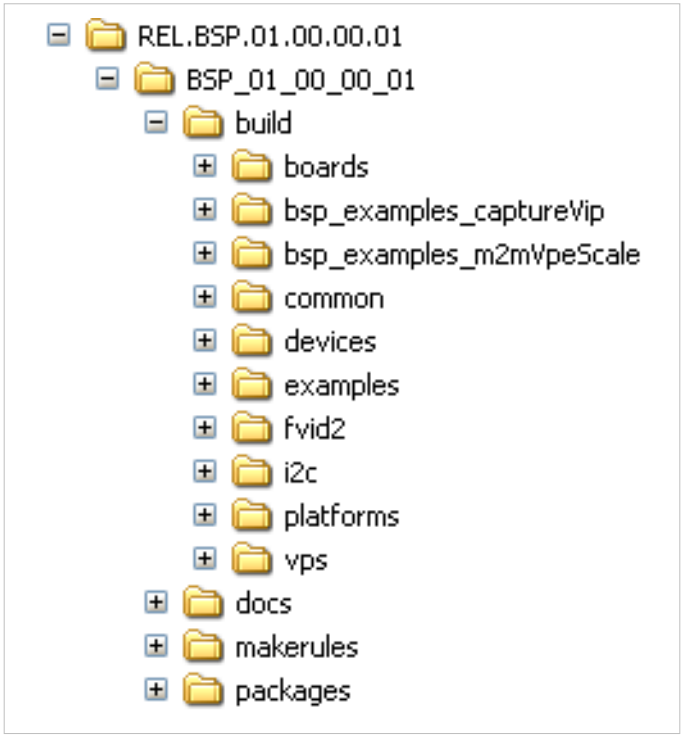
Top Level

On successful installation of BSP source code, in the installed directory following folders would be created. Lets consider 01_00_00_01 version release as an example.



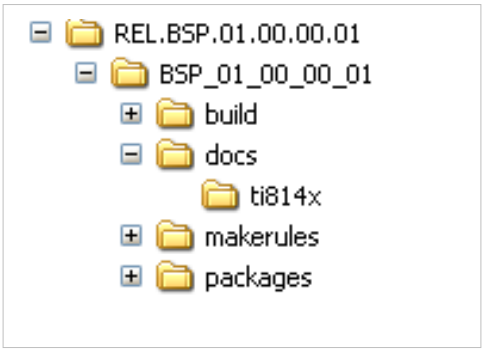
Build

Build directory essentially holds all the generated binaries and libraries. For each sample application that comes with this BSP 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.



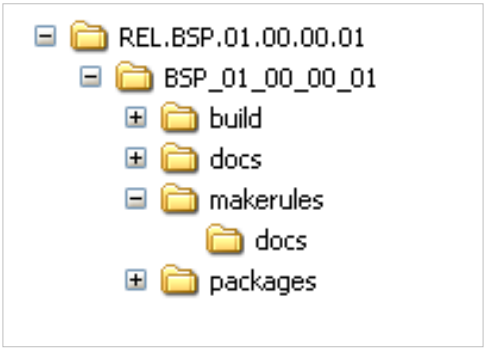
Docs

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 ti816x, ti814x, ti813x) 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 BSP drivers, sample applications and other utilities. The docs folder under makerules folder contain makerules_spec.doc document that elaborates on the make files used.

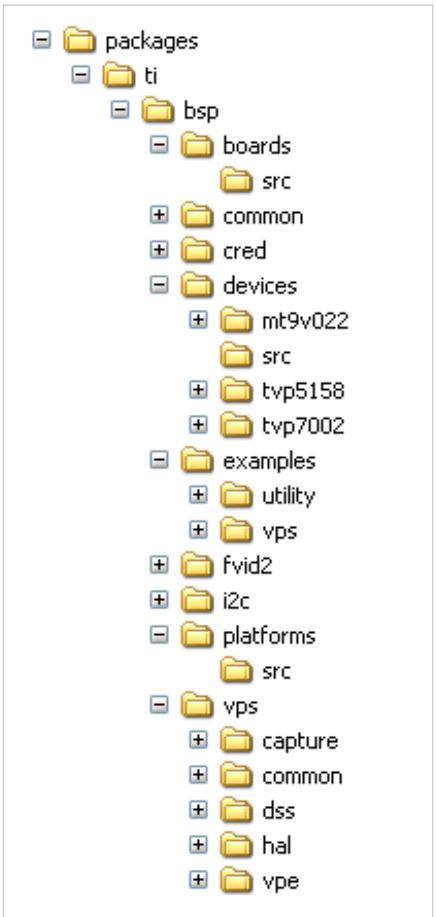


Packages

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

Packages

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

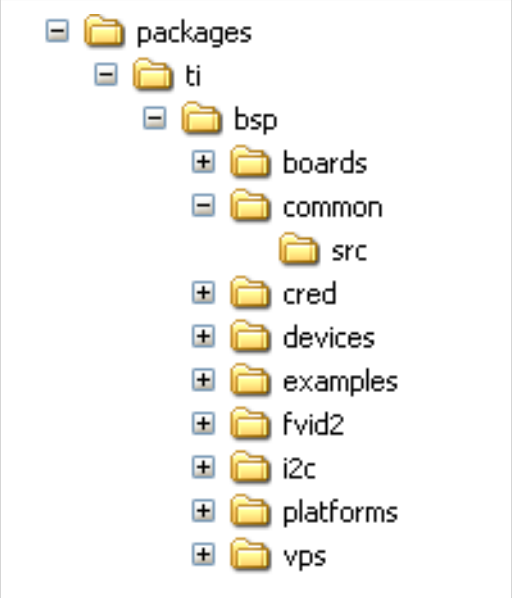


Boards

Contains list of on-board devices/daughter card such as Video Surveillance (VS), Video Conferencing (VC), Vision Application board, Custom board, etc. Contains routines to auto-detect the daughter card/board type. Any special programming needed by the board is also added here.

Common

Contains routines to log TRACE, OS abstraction layer, Heap management like BSS memory and VPDMA descriptor memory, and common utility functions like link lists, queue, etc.



CRED

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.

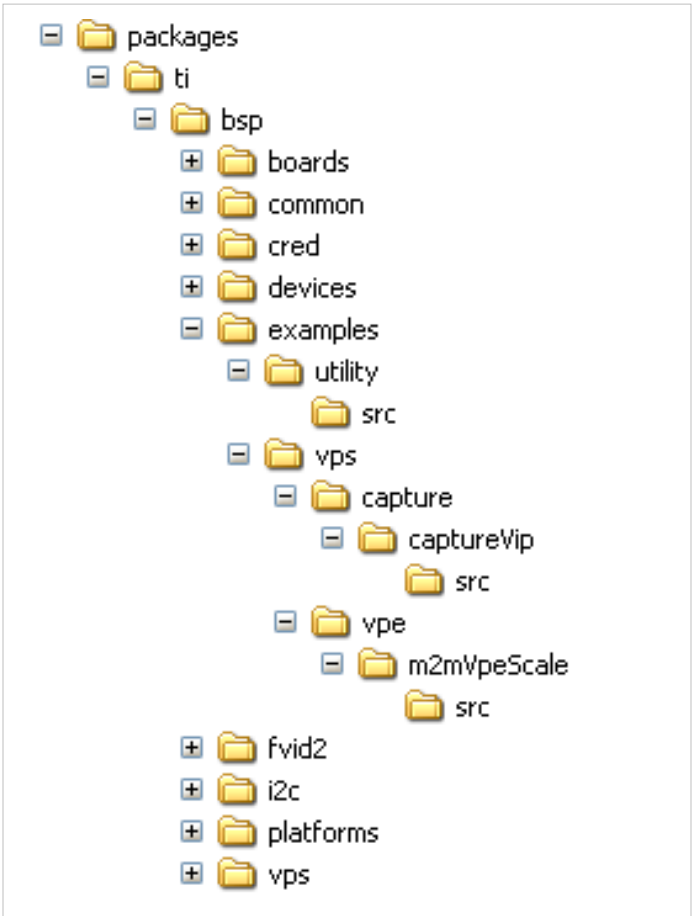
Devices

Contains driver / function implementation to control on-board encoders, decoders, sensors, other devices such as TVP7002, TVP5158, MT9V022 sensor, 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 bspdrv_tvp7002.h at \packages\ti\bsp\devices\tvp7002 - Defines the interface exposed by the TVP7002 decoder driver, typically initialization and de-initialization function called by device initialization. \devices\src\bsp_device.c Encoder / decoder device driver implementation could be found at - \packages\ti\bsp\devices\tvp7002\src

Examples

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



Utility

This folder contains application level utility functions which could be used as is or with minimal modification. This folder also contains routines for TRACE log.

Vps

This folder contains sample applications for various VPS drivers such as VIP, VPE and DSS drivers.

Capture

This folder contains sample applications for capture (VIP) drivers.

VPE

This folder contains sample applications for various video processing elements (memory2memory) drivers.

Fvid2

Contains the FVID2 API framework and driver level interface files.

I2C

Contains the i2c driver files, that could be used to read / write into any of the video on-board devices such as sii9022a, TVP7002, IO Expanders, or sensors such as MT9V022, etc. The interface exposed by I2C driver is available at \packages\ti\bsp\i2c\src\bsp_i2cdrv.h. The source I2C driver is available at \packages\ti\bsp\i2c\src\bsp_i2cdrv.c

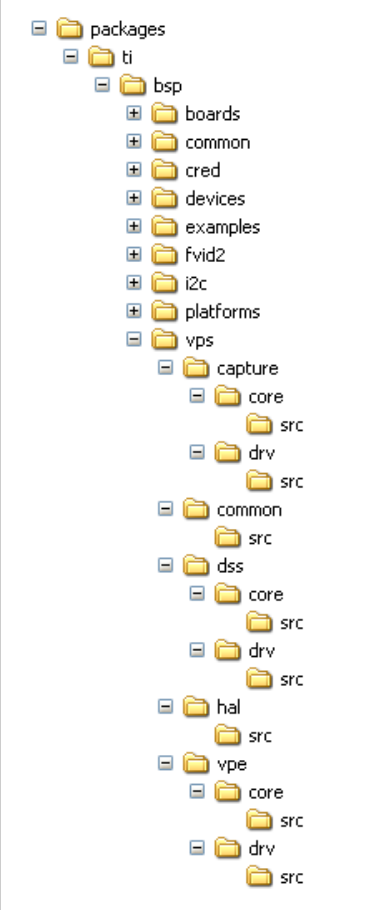
Platforms

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 BSP initialization, the platform functionality would be initialized

VPS

Is the root folder that contains all the driver implementations for Capture (VIP), VPE and DSS. This folder also contains routines for VIP, VPE and DSS core drivers and HAL layer routines. The BSP drivers are segregated into capture, dss and vpe drivers. Each of the sub-folder under driver folder implements drivers and core routines. Each of the sub-folder have consistent sub-folder directory structure.

The makefile to build the entire driver is also contained here. This makefile allows the user to build exclusively the HAL package, Capture (VIP) driver, VPE driver and DSS driver package.



Capture

This folder contains the capture (VIP) specific core and driver implementations.

VPE

This folder contains the video processing elements (memory2memory) specific core and driver implementations.

DSS

This folder contains the display sub-system specific core and driver implementations.

core

The common functional block / paths that could potentially be used by the capture/vpe/dss driver, is configured / managed by common software entity called “core”.

The implementation of capture (VIP) core is at \packages\ti\bsp\vps\capture\core\src\ and the interface files of core is at \packages\ti\bsp\vps\capture\core\

The implementation of vpe core is at \packages\ti\bsp\vps\vpe\core\src\ and the interface files of core is at \packages\ti\bsp\vps\vpe\core\

The implementation of dss core is at \packages\ti\bsp\vps\dss\core\src\ and the interface files of core is at \packages\ti\bsp\vps\dss\core\

drv

The common driver implementation for capture/vpe/dss.

The source implementation for capture driver is at \packages\ti\bsp\vps\capture\drv\src\ and the interface can be found at \packages\ti\bsp\vps\capture\drv\

The source implementation for vpe driver is at \packages\ti\bsp\vps\vpe\drv\src\ and the interface can be found at \packages\ti\bsp\vps\vpe\drv\

The source implementation for dss drivers is at \packages\ti\bsp\vps\dss\drv\src\ and the interface can be found at \packages\ti\bsp\vps\dss\drv\

Common

The common functionality such as CORE and HAL initializations, event management, resource management is implemented in this folder. The files under \packages\ti\bsp\vps\common\ provides the interfaces that could be used by the drivers. The folder \packages\ti\bsp\vps\common\src\ is the place holder for the implementation.

HAL

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

Vayu-Overview

TDA2XX VPS Hardware Introduction

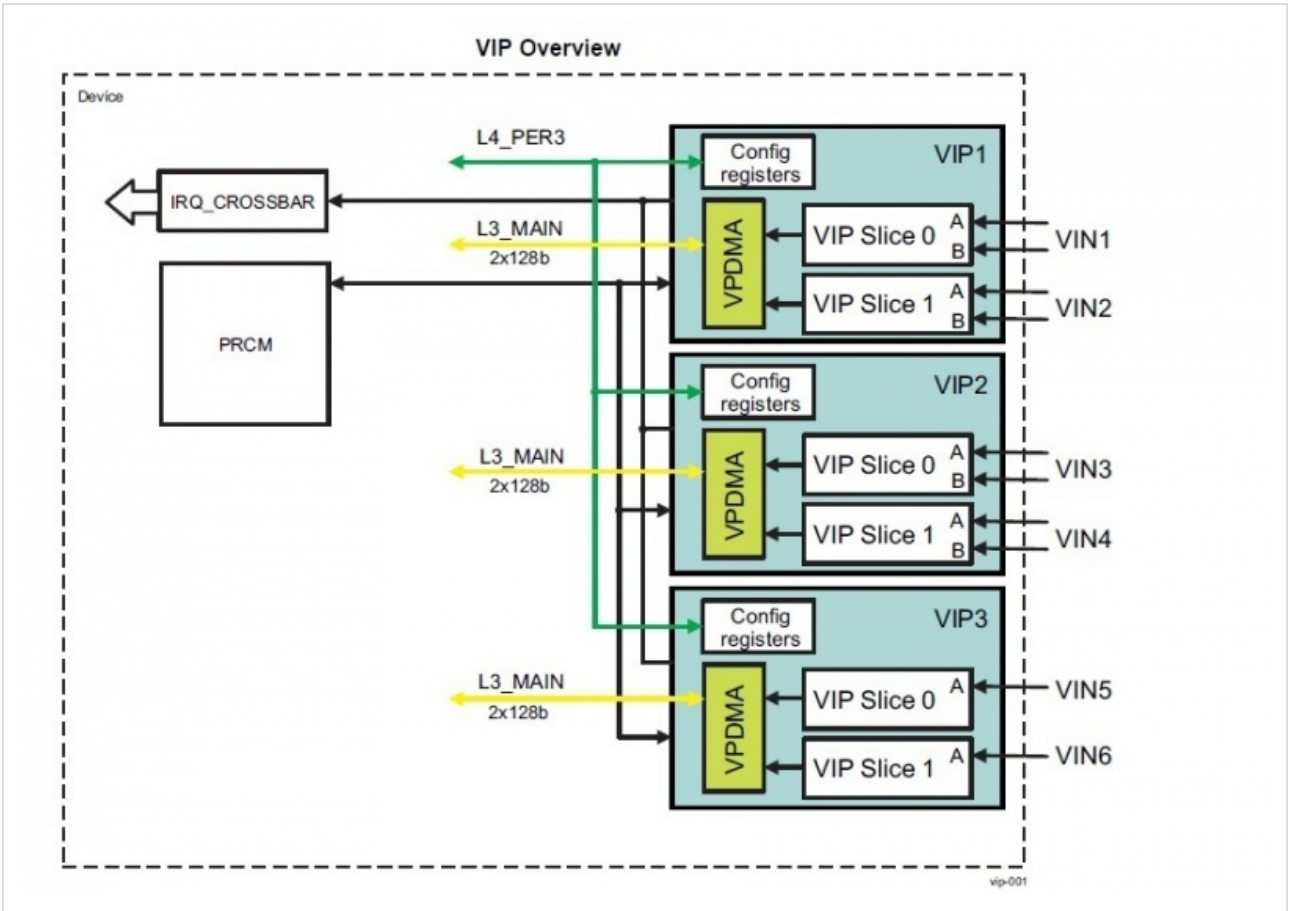
Video Processing system comprises of VIP for video capture, VPE for processing on video data, and DSS for video display output.

Video Input Port (VIP)

Video Input Port (VIP) is the video capture interface on TDA2XX .The VIP module provides video capture functions for the device. VIP incorporates a multi-channel raw video parser, various video processing blocks, and a flexible DMA engine to store incoming video in various formats. The device uses three instantiations of the VIP module giving the ability of capturing up to six video streams.

There are 3 instances of VIP in TDA2XX . Each instance of VIP comprises of 2 slices and 1 VPDMA. Each slice in turn has 2 input ports Port A and Port B. Port A can be configured for 24bit (YUV444/RGB888), 16bit (YUV422), 8bit time interleaved (YUV422). Port B is only 8bit time interleaved (YUV422).VIP3 does not use Port B of both slices and Port A supports only 16bit interface. This implies that on Vayu 12 8bit capture is possible using the 3 VIPs, 6slices and 2 ports (A and B).

Figure shows a block diagram with the VIP module within the device.



Supported Features

- Each video input Port A port can be operated as clock independent input channels (with interleaved or separated Y/C data input). Embedded sync and external sync modes are supported for all input configurations.
- Support for a single external asynchronous pixel clock, up to 165 Mhz per port.
- Pixel Clock Input Domain Port A supports up to one 24-bit input data bus, including BT.1120 style embedded sync for 16 and 24 bit data.
- Support for embedded (BT.656/BT.1120 16/24b, BT.656 8b) or discrete (BT.601 style) sync
- Embedded Sync data interface mode (can support multiplexed sources)
- Discrete Sync data interface mode (only supports single source/non-multiplexed inputs)
- 24b data input plus discrete syncs, can be configured to include:
  - 8b YUV422 (Y and U/V time interleaved)
  - 16b YUV422 (CbY and CrY time interleaved)
  - 24b YUV444
  - 16b RGB656
  - 24b RGB888
  - 8b RAW Capture
  - 16b RAW Capture
  - 24b RAW Capture
- Discrete sync modes include:
  - VSYNC + HSYNC (FID determined by FID signal pin or HSYNC/VSYNC skew)
  - VSYNC + ACTVID + FID
  - VBLANK + ACTVID (ACTVID toggles in VBLANK) + FID
  - VBLANK + ACTVID (no ACTVID toggles in VBLANK) + FID
- Multichannel parser (embedded syncs only)
  - Pixel (2x or 4x) or Line multiplexed modes supported
  - Performs demultiplexing and basic error checking
  - Supports maximum of 9 channels in Line Mux (8 normal + 1 split line)
- Ancillary data capture support
  - For 16b or 24b input, ancillary data may be extracted from any single channel
  - For 8b time interleaved input, ancillary data can be chosen from the Luma channel, the Chroma channel, or both channels
  - Horizontal blanking interval data capture only supported when using discrete syncs (VSYNC + HSYNC or VSYNC + HBLANK)
  - Ancillary data extraction supported on multichannel capture as well as single source streams
- Format conversion and scaling
  - Programmable color space conversion
  - 422 => 444 conversion
  - 444 => 422 conversion
  - 422 => 420 conversion
  - YUV444 Source:

```
       YUV444 => YUV444, YUV444 => RGB888, YUV444 => YUV422,
YUV444 => YUV420,
```

  - RGB888 Source:

```
       RGB888 => RGB888, RGB888 => YUV444, RGB888 => YUV422,
RGB888 => YUV420
```

- YUV422 Source:

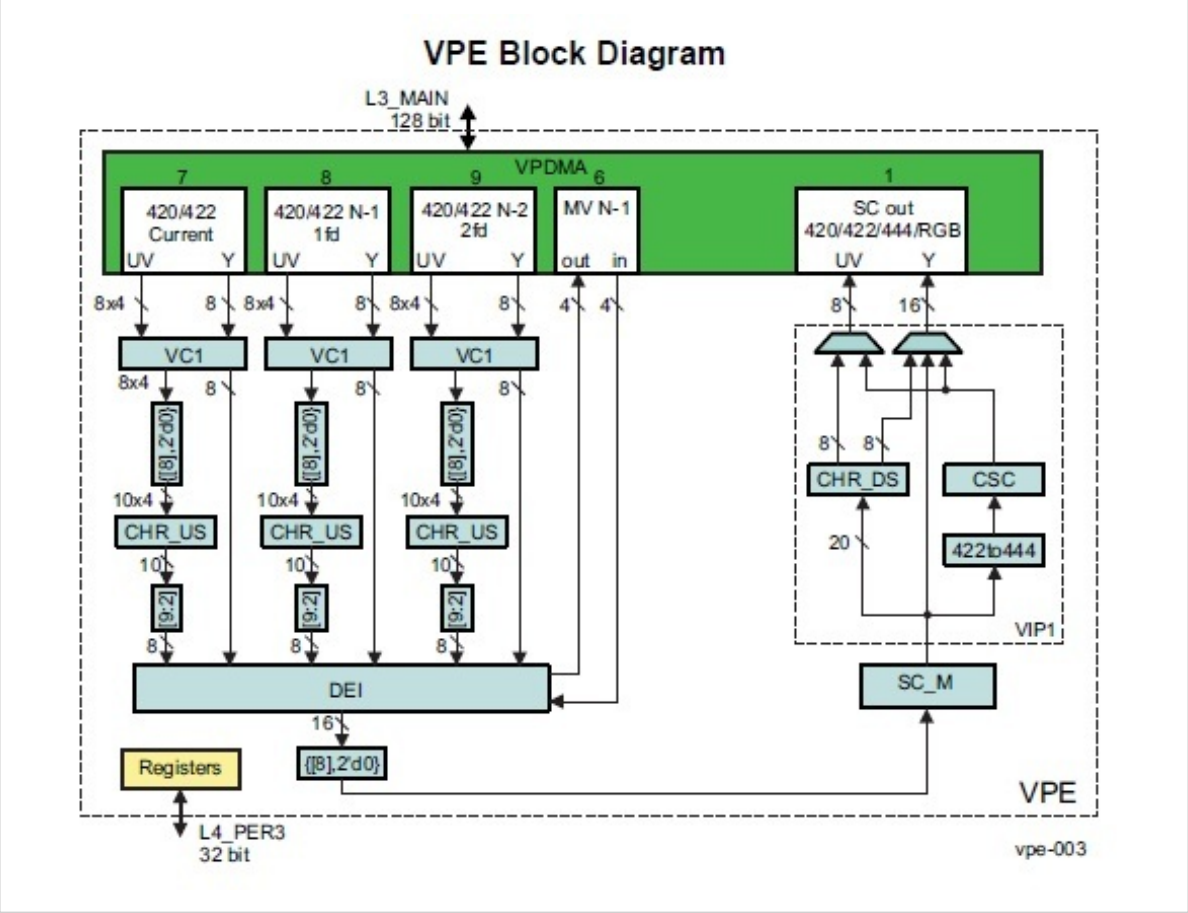
```
        YUV422 => YUV422, YUV422 => YUV420, YUV422 => YUV444,
YUV422 => RGB888
```

  - Supports RAW to RAW (no processing)
  - Scaling and format conversions do not work for multiplexed input
- Support for 1080P input with scaling and chroma up/down sampling (1920 wide line buffers)
- VPDMA can transfer the capture data into the buffer in external memory or OCMC memory.
  - VPDMA output buffer size restriction feature ensures that writes not exceed allocated memory buffer size
  - Support for Tiled (2D) and raster addressing without bandwidth penalty
  - Dual clients per channel allows for capture of scaled and non-scaled versions of the data stream (non-multiplexed node only)
  - Start on new frame capability
  - Interrupt every X number of frames
  - Interrupt every X lines (synced to frame start)

Video Processing Engine (VPE)

VPE Features

Figure shows a VPE block diagram.



- Supports memory to memory operations only.
- VPE consist of a single memory to memory path which can perform the following operations:
  - Read of raster or tiled YUV420 coplanar, YUV422 coplanar or YUV422 interleaved video

- Deinterlacing of the input video using a 4 field motion based algorithm
- Scaling of the input video up to 1080p (1920x1080) resolution
- Write of the resulting video in YUV420 coplanar (raster or tiled), YUV422 coplanar (raster or tiled),YUV422 interleaved (raster or tiled), YUV444 single plane (raster only) or RGB888 (raster only)
- Deinterlacing up to two 1080i video sources.
- The single data path performs operations in the following order
  - Chroma Upsampling from 420 to 422 (if needed)
  - Deinterlacing of 422 video from interlaced to progressive (if needed)
  - Scaling of 422 video after deinterlace
  - Conversion of 422 video to 420, 444 or RGB (if needed)
- VC-1 Range Mapping and Range Reduction support on input video before Chroma Upsampling (if needed)

- Chroma Upsampling Features
  - 4 line Catmull-Rom based implementation
  - Programmable coefficients for interlaced or progressive conversion. Separate coefficients can be provided for top and bottom fields
- Deinterlacer Features
  - 8-bit, YCbCr 4:2:2
  - Motion-adaptive deinterlacing (MDT)
    - Motion detection is based on Luma only
    - 4-field data is used
    - Motion values adaptive to the frequency of luma texture
  - Edge-Directed Interpolation (EDI)
    - Edge detection using luma pixels in a 2x7 window
    - Seven edge vectors: -1.5, -1, -0.5, 0, 0.5, 1, 1.5
    - Edge-directed chroma interpolation
    - Soft-switch between edge directed interpolation and vertical interpolation depending on the confidence factor
  - Film Mode Detection (FMD)
    - 3-2 pull down detection
    - 2-2 pull down detection
    - Hysteresis controls how fast FMD can enter/exit film mode (software function)
    - Bad Edit Detection (BED)
  - Progessive Input
    - For Progressive Input, the module passes input to output. No internal processing is performed.This is essentially a bypass mode.
  - Interlace Bypass
    - For Interlace Input, the module can pass the inputs data directly to the outputs in a bypass configuration. No internal processing is performed
- Scaler Features
  - Vertical and horizontal up/down scaling
    - Polyphase filter upscaling
  - Running average vertical down scaling for memory optimization
  - Decimation and polyphase filtering for horizontal scaling
  - Non-linear scaling for stretched/compressed left and right sides
  - Input image trimmer for pan/scan support

    - Pre-scaling peaking filter for enhanced sharpness
    - Scale field as frame
    - Interlacing of scaled output
    - Full 1080p input and output support
    - YCbCr422 input and output
    - Minimum horizontal scaling ratio = 1/8x
    - Maximum horizontal scaling ratio – limited by output line buffer (2014 pixels)
    - Scaling filter Coefficient memory download
- Chroma Downsampler Features
    - Simple two-line averager capable of converting from YUV422 to YUV420 space
- 422 to 444 Features
    - Catmull-Rom based filter
    - 4 pixel, fixed coefficient
- Color Space Converter Features
    - Fully programmable 3x3 matrix multiplier with offset control

Display Sub System (DSS)

The display subsystem provides the logic to display a video frame from the memory frame buffer on a liquid-crystal display (LCD) panel or TV set.

The display subsystem can display different pictures simultaneously by using three LCD outputs (LCD1, LCD2, LCD3), in addition to the TV output.

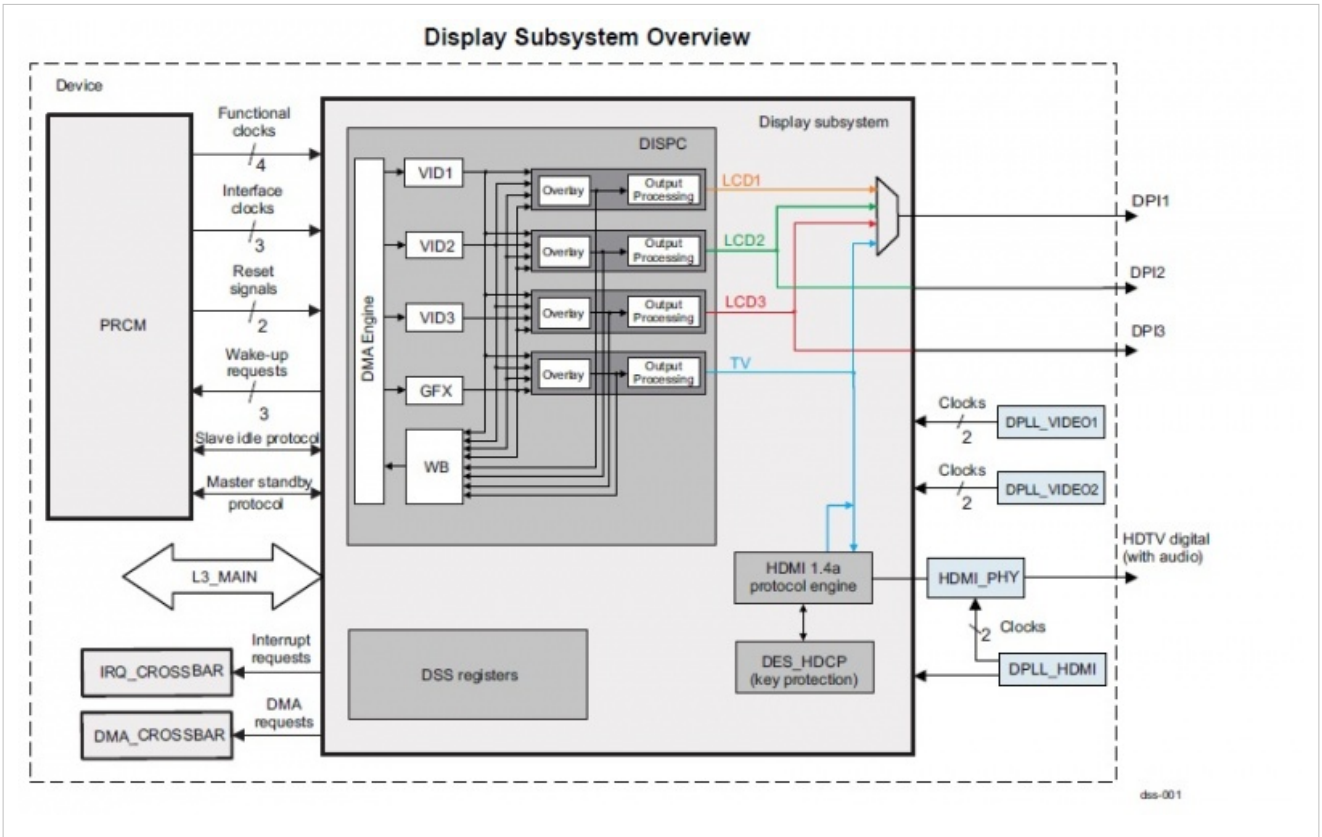
All three LCD outputs can use separate parallel CMOS output interfaces (DPI) (MIPI DPI 2.0, BT-656, or BT-1120).

- DPI composite signal (DPI1, DPI2, DPI3)

The TV output can be one of the following:

- High-definition multimedia interface (HDMI)
- DPI composite signal (DPI1)

Figure shows a block diagram with the DSS module within the device.



The modules integrated in the display subsystem are:

- Display controller (DISPC):
  - One direct memory access (DMA) engine
  - Three LCD outputs and one TV output, each with a dedicated overlay manager
  - One graphics pipeline (GFX), three video pipelines, and one write-back pipeline
- HDMI protocol engine:
  - HDMI 1.4 support at 1080p, 60Hz (including 3D frame-packing support)
  - 36-bit RGB color
  - HDCP 1.4 key protection
  - Deep color mode support (10-bit/12-bit for 148.5-MHz pixel clock)

The necessary phase-locked loops (PLLs) with their control modules and the physical layer for HDMI (PHY) are outside the display subsystem. There are three PLLs allowing independant display output at diferent frequencies. The PLLs and PHY are as follows:

- DPLL_HDMI/HDMI_PHY
- DPLL_VIDEO1
- DPLL_VIDEO2

To ensure efficient bandwidth, the display subsystem integrates a connection between the level 3 (L3_MAIN) interconnect and the DISPC to exchange data with synchronous dynamic random access memory (SDRAM) and memory using the DISPC DMA engine. This connection is also used for configuration.

TDA3XX-Overview

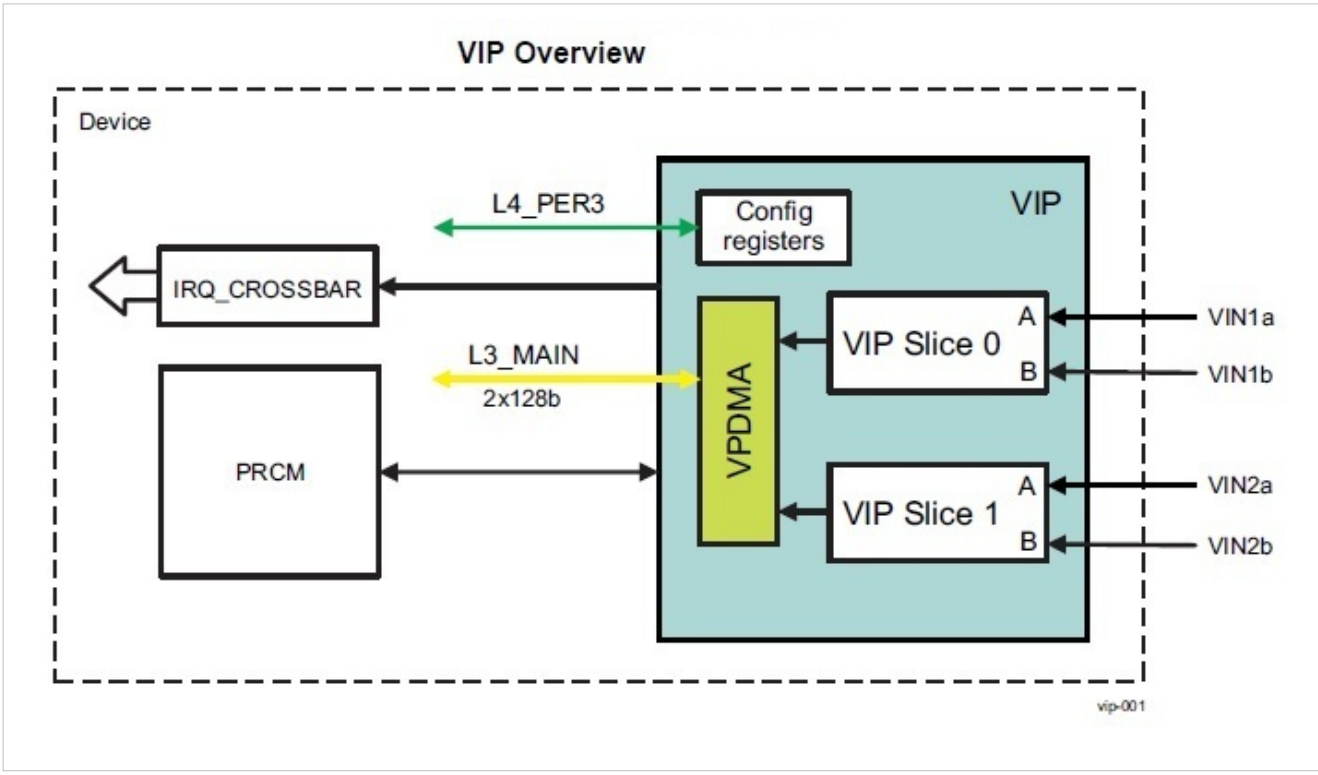
TDA3XX VPS Hardware Introduction

Video Processing system comprises of VIP for video capture,DSS for video display output and ISS for image processing.

Video Input Port (VIP)

The VIP module provides video capture functions for the device. VIP incorporates a multi-channel raw video parser, various video processing blocks, and a flexible DMA engine to store incoming video in various formats. The device uses a single instantiation of the VIP module giving the ability of capturing up to two video streams.

Figure shows a block diagram with the VIP module within the device.



Supported Features

- Two independently configurable external video input capture slices each of which has two video input ports, Port A and Port B, where Port A can be configured as a 24/16/8 bit port (24-bit supported on slice 0 only; slice 1 can be 16/8 bit), and Port B is a fixed 8bit port.
- Each video input Port A port can be operated as clock independent input channels (with interleaved or separated Y/C data input). Embedded sync and external sync modes are supported for all input configurations.
- Support for a single external asynchronous pixel clock, up to 165 Mhz per port.
- Pixel Clock Input Domain Port A supports up to one 24-bit input data bus, including BT.1120 style embedded sync for 16 and 24 bit data.
- Support for embedded (BT.656/BT.1120 16/24b, BT.656 8b) or discrete (BT.601 style) sync
- Embedded Sync data interface mode (can support multiplexed sources)
- Discrete Sync data interface mode (only supports single source/non-multiplexed inputs)
- 24b data input plus discrete syncs, can be configured to include:

    - 8b YUV422 (Y and U/V time interleaved)
    - 16b YUV422 (CbY and CrY time interleaved)
    - 24b YUV444
    - 16b RGB656
    - 24b RGB888
    - 8b RAW Capture
    - 16b RAW Capture
    - 24b RAW Capture
- Discrete sync modes include:
    - VSYNC + HSYNC (FID determined by FID signal pin or HSYNC/VSYNC skew)
    - VSYNC + ACTVID + FID
    - VBLANK + ACTVID (ACTVID toggles in VBLANK) + FID
    - VBLANK + ACTVID (no ACTVID toggles in VBLANK) + FID
- Multichannel parser (embedded syncs only)
    - Pixel (2x or 4x) or Line multiplexed modes supported
    - Performs demultiplexing and basic error checking
    - Supports maximum of 9 channels in Line Mux (8 normal + 1 split line)
- Ancillary data capture support
    - For 16b or 24b input, ancillary data may be extracted from any single channel
    - For 8b time interleaved input, ancillary data can be chosen from the Luma channel, the Chroma channel, or both channels
    - Horizontal blanking interval data capture only supported when using discrete syncs (VSYNC + HSYNC or VSYNC + HBLANK)
    - Ancillary data extraction supported on multichannel capture as well as single source streams
- Format conversion and scaling
    - Programmable color space conversion
    - 422 => 444 conversion
    - 444 => 422 conversion
    - 422 => 420 conversion
    - YUV444 Source:

```
       YUV444 => YUV444, YUV444 => RGB888, YUV444 => YUV422,
YUV444 => YUV420,
```

    - RGB888 Source:

```
       RGB888 => RGB888, RGB888 => YUV444, RGB888 => YUV422,
RGB888 => YUV420
```

    - YUV422 Source:

```
       YUV422 => YUV422, YUV422 => YUV420, YUV422 => YUV444,
YUV422 => RGB888
```

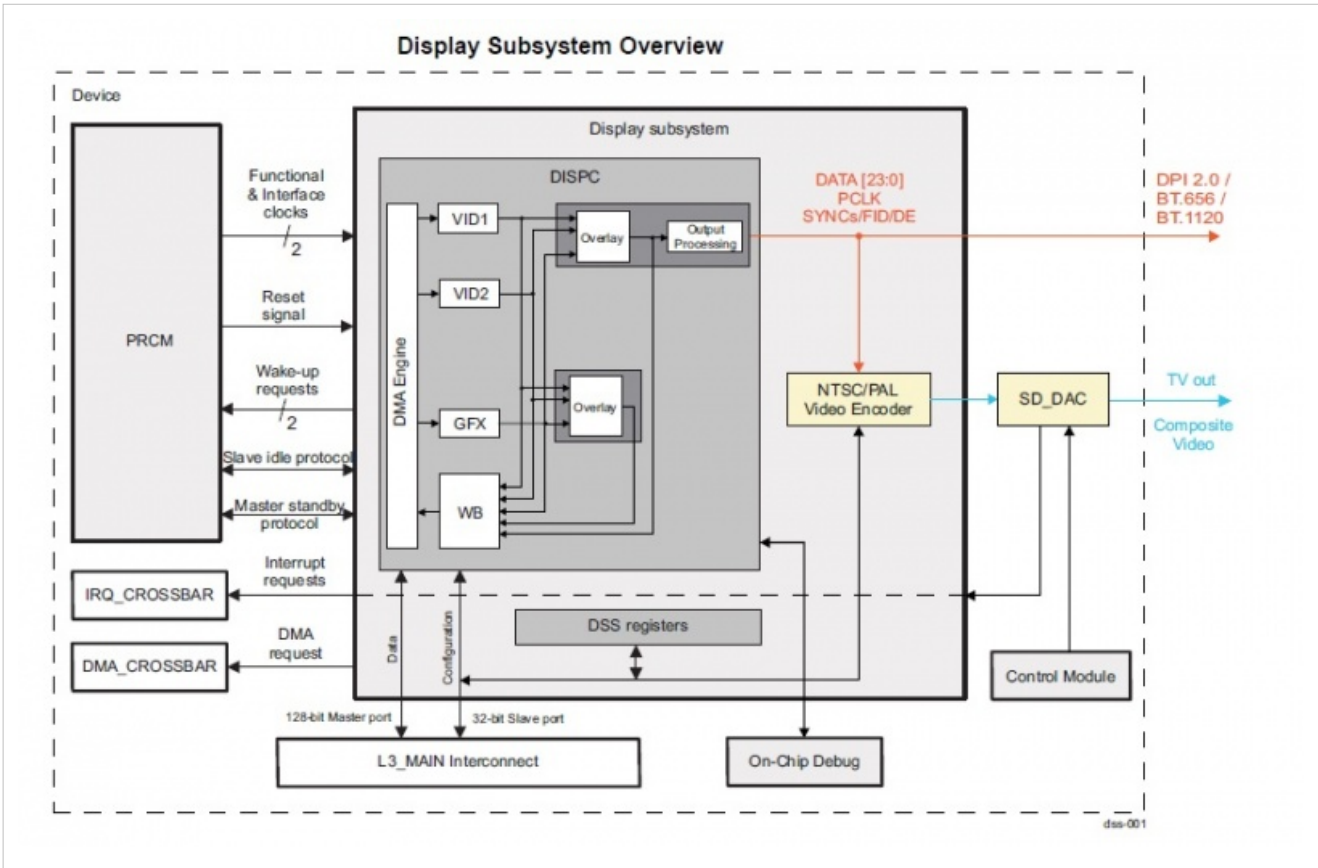
    - Supports RAW to RAW (no processing)
    - Scaling and format conversions do not work for multiplexed input
- Support for 1080P input with scaling and chroma up/down sampling (1920 wide line buffers)
- VPDMA can transfer the capture data into the buffer in external memory or OCMC memory.
    - VPDMA output buffer size restriction feature ensures that writes not exceed allocated memory buffer size

- Support for Tiled (2D) and raster addressing without bandwidth penalty
- Dual clients per channel allows for capture of scaled and non-scaled versions of the data stream (non-multiplexed node only)
- Start on new frame capability
- Interrupt every X number of frames
- Interrupt every X lines (synced to frame start)

Display Sub System (DSS)

The Display Subsystem (DSS) provides the logic to interface display peripherals. DSS integrates a DMA engine as part of DISPC module, which allows direct access to the memory frame buffer. Various pixel processing capabilities are supported, such as: color space conversion, filtering, scaling, blending, color keying, etc.

Figure shows a block diagram with the DSS module within the device.



The supported display interfaces are:

- One parallel CMOS output, which can be used for MIPI® DPI 2.0, or BT-656 or BT-1120.
- One TV output, which is connected to the internal Video Encoder module (VENC). The VENC drives a single video digital-to-analog converter (SD_DAC) supporting composite video mode.

The modules integrated in the display subsystem are:

- Display controller (DISPC), with the following main features
  - One direct memory access (DMA) engine
  - One graphics pipeline (GFX), two video pipelines (VID1 and VID2), and one write-back pipeline(WB)
  - Two overlay managers
  - Active Matrix color support for 12/16/18/24-bit (truncated or dithered encoded pixel values)
  - One Video Port (VP) with programmable timing generator to support:

    - DPI: up to 165 MHz pixel clock video formats defined in CEA-861-E and VESA DMT standards
    - VENC: NTSC/PAL standards with 60Hz/50Hz refresh rates
  - Supported maximum FrameBuffer width of 4096 for all pixel formats
  - Configurable output mode: progressive or interlaced
  - Selection between RGB and YUV422 output pixel formats (YUV4:2:2 only available when BT-656 or BT-1120 output mode is enabled)
- Video Encoder (VENC) with 10-bit standard definition video DAC (SD_DAC).

DSS provides two interfaces to L3_MAIN interconnect

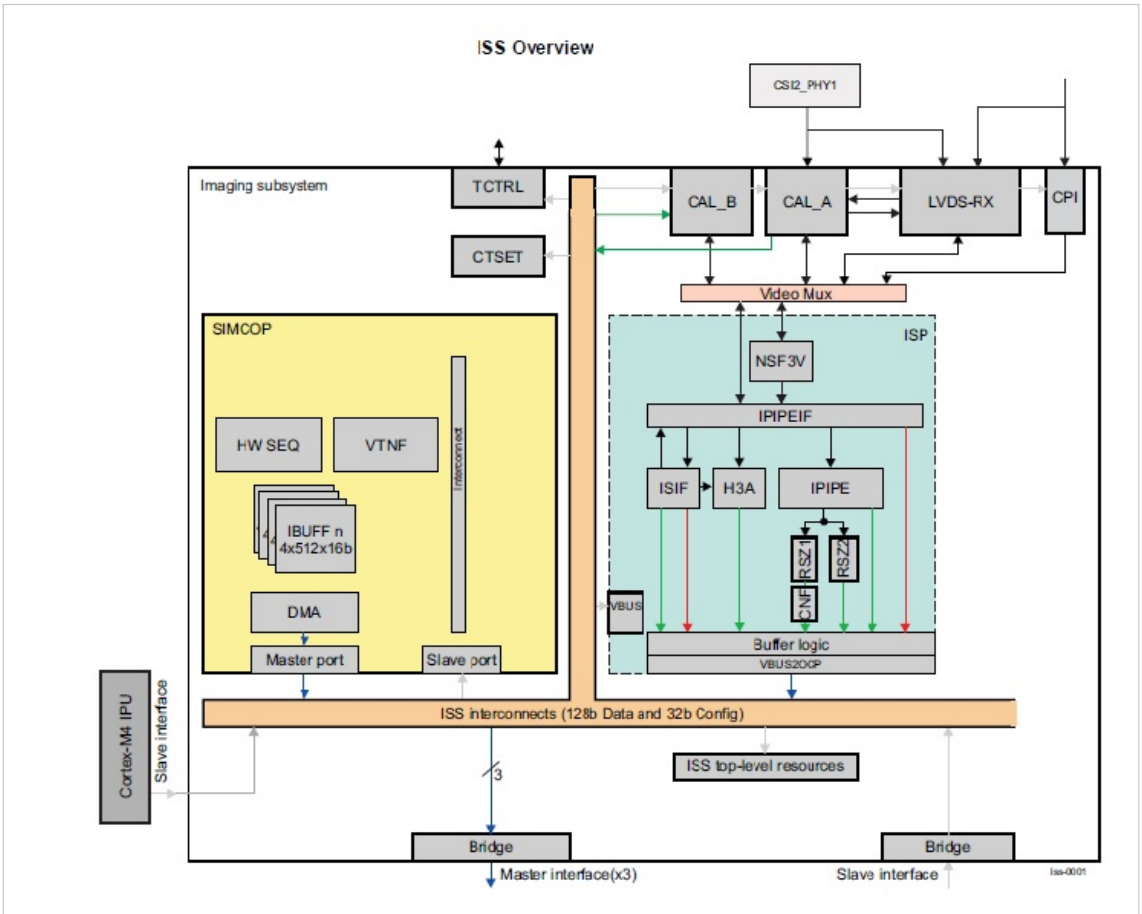
- One 128-bit master port (with MFLAG support). The DMA engine in DISPC uses this single master to read/write data from/to device system memory.
- One 32-bit slave port. Used for registers configuration. It is further connected internally to DISPC and VENC modules.

Imaging Sub System (ISS)

The imaging subsystem (ISS) deals with the processing of the pixel data coming from an external image sensor or data from memory (image format encoding and decoding can be done to and from memory).With its subparts, such as interfaces and interconnects, image signal processor (ISP), and still image coprocessor (SIMCOP), the ISS is a key component for the following use cases:

- Rear View Camera
- Front View Stereo Camera
- Surround View Camera

Figure shows an overview of the ISS.



The direction of the arrows shows the command flow direction from the master (initiator) to the slave (target). The following color conventions are used for the connections:

- Blue: Bidirectional, 128-bit-wide interface data connection
- Green: Write (ISS → system memory) data connection. Either 64-bit interface, 128-bit interface, or 32-bit MTC (inside ISP).
- Red: Read (system memory to ISS) data connection. 128-bit interface port or 32-bit MTC (inside ISP).
- Gray: 32-bit interface configuration connection
- Solid black: Video port and camera interface related signals
- Dotted black: Data flow stall control signal. Used to slow down ISP for memory-to-memory operation.

The ISS is mainly composed of CAL_A, CAL_B, LVDS-RX camera interfaces, a parallel interface (CPI),an ISP, and a block-based imaging accelerator (SIMCOP).

The ISS is designed to reach high throughput and low latency with large image sensors. In highperformance mode, the ISP supports a pixel throughput of 212.8 MPix/s.

The ISS is tightly coupled with a low-interrupt latency microprocessor subsystem (Cortex®-M4 IPU) that runs a real-time operating system (OS) to reach optimal performance. Mainly, the Cortex-M4 IPU can quickly change the ISS configuration during frame blanking periods and run some sequencing tasks. It can also be configured using the main MPU subsystem (Cortex®-A15 MPU) or the system DMA controller (DMA_SYSTEM). Typically, the Cortex-A15 MPU can run some less latency-sensitive tasks, and the DMA_SYSTEM can be used to transfer large configuration tables used by the ISS (for example, Gamma tables and iMX software).

The ISS targets the following major use cases:

- Video / Preview
  - Up to 1080p60
  - Up to 2x 1080p30
- Stereo Video / Preview
  - Up to 2x1080p30
- Zero or negative shutter lag still image capture
- Multi-camera use cases
  - Up to 4 simultaneous cameras. ISP time is shared.

The ISS offers the following features:

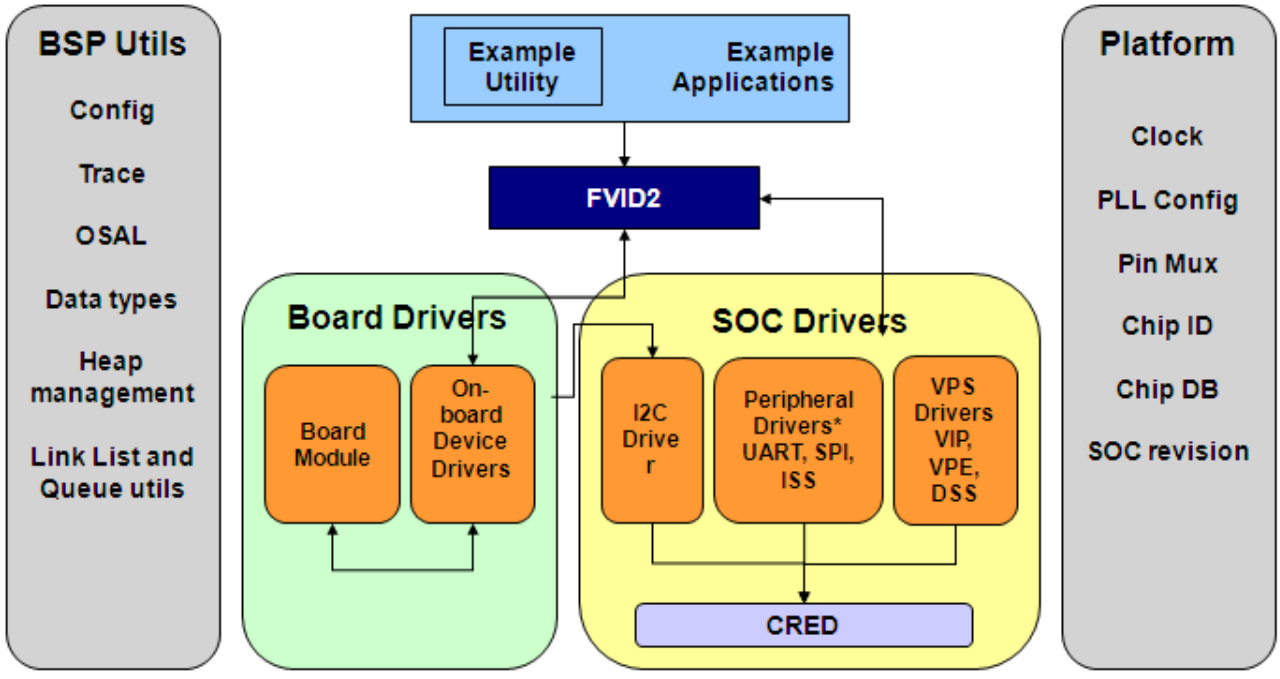
- ISS interfaces:
  - CAL_A camera interfaces with external D-PHY and write DMA
  - CAL_B camera interfaces with read DMA
  - Parallel interface (CPI) (16 bits wide, with up to 212.8 MPix/s throughput, and supporting BT656,SYNC modes)
  - LVDS receiver
  - 128-bit-wide data interface to the level 3 (L3_MAIN) interconnect
  - The ISS relies on the centralized memory management unit (MMU).
- ISP:
  - On-the-fly or memory-to-memory processing
  - Up to 212.8 MPix/s throughput
  - Statistic data collection
  - Image pipe interface front-end raw data processing
  - High-ISO video noise filtering (NSF3V)
  - RGB and YUV data processing through ISIF and IPIPE
  - Two image continuous real-time resizers

- Chroma noise filter
- SIMCOP:
  - Memory-to-memory operation
  - Temporal video noise filter (VTNF)
  - Direct memory access (DMA) controller
  - Hardware sequencer
- ISS interconnect:
  - 128-bit-wide network for image data (full speed)
  - 32-bit-wide network for configuration (half speed)
  - Hard and soft real-time data flows
  - CAL_A, CAL_B, ISP, and SIMCOP data flow management.

BSP-Software Overview

BSP Driver Introduction

BIOS Support Package (BSP) block diagram is as given below:



BSP Software Architechture

The Video Processing System (VPS) drivers could be broadly categorized as:

- Capture Driver
- Video Processing Elements (memory to memory) Driver
- Display Driver

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.
- 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 (VIP1, VIP2, VIP3), multi-slice (S0, S1), and multi-port capture (Port A, Port B), with ability to configure each instance, slice, port, independently.
- Capture drivers are supported on:
  - Interfaces : FVID2
  - OS :BIOS6
  - Processor : Ducati M3 and M4

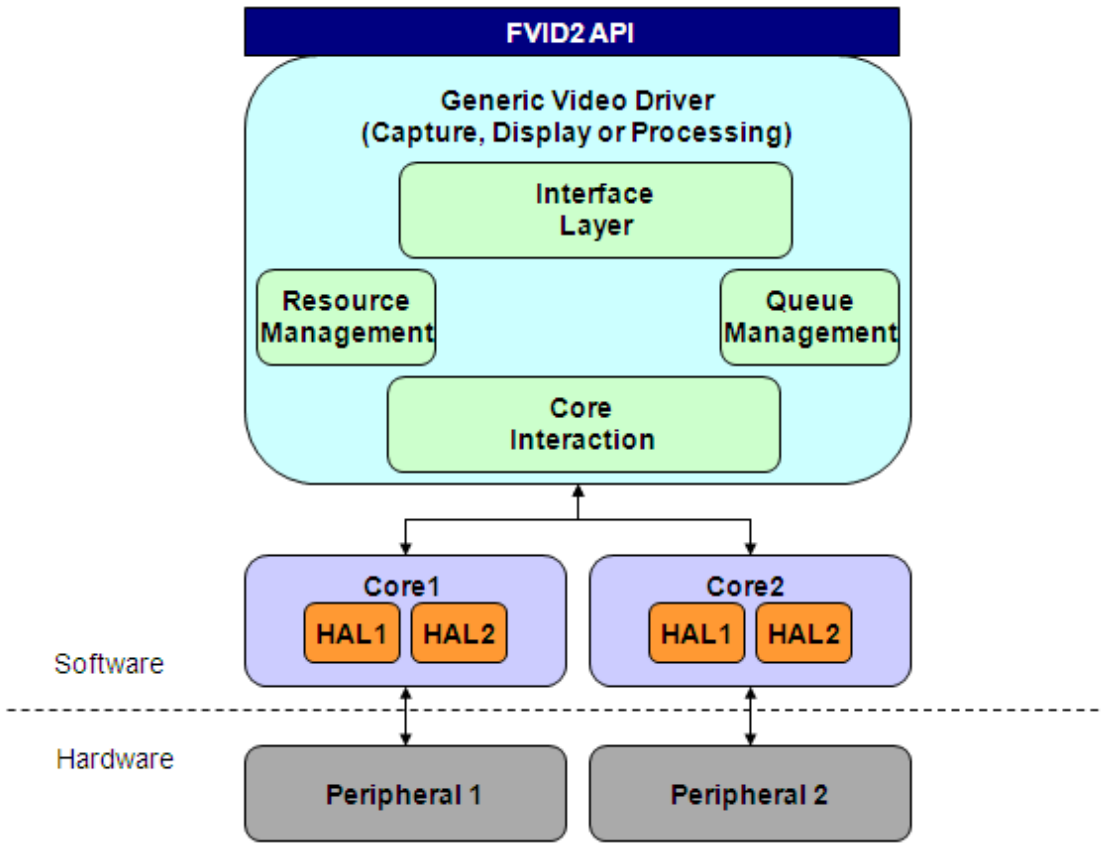
Memory (M2M) Driver

To be Done.

Display Driver

To be Done.

BSP Driver Stack



BSP Driver Stack

Different layers or components/modules of BSP are:

- **FVID2 Layer**- FVID2 API interface is used for interaction between application and the BSP driver.
- **Generic Video Driver layer**- The driver layer handles the application-driver interface through the FVID2 APIs. The driver layer is also responsible for queue and de-queue of buffers, does the resource management, handles the driver-core interactions.

There are multiple driver possible because of various paths. Different path may use same IP like VIP1 Port A and VIP1 Port B path in Capture driver. It means only one of driver could be active at any point of time. Resource manager handles allocation resources to different drivers. The resource manager also handles the VPDMA list management as mentioned in the Core Layer.

Event manager parses the interrupt status register to figure out different kind of interrupts and propagates to different modules of BSP stack.

- **Core layer**- Core Layer invokes the appropriate HAL APIs. It is also responsible for creation of data transfer 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 Core specific DMA engine for data transfer to external memory/OCMC buffer or for setting different configuration like height/width of frame, scalar coefficients, etc.

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. The list manager is part of the resource manager.

- **HAL layer**- Hardware Abstraction layer – as name suggest – it abstracts multiple IPs of BSP and provides interfaces for upper layer of stacks.
- **HW layer**- It is TDA2SEDX VIP/VPE/DSS H/w layer.

UserGuideFVID2 Vayu

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.

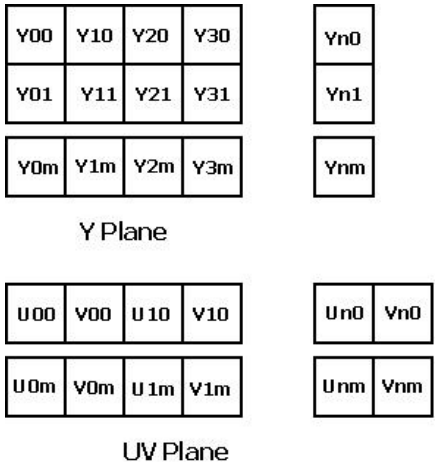
Warning
Underlying drivers catering to FVID2 interfaces may decide to expose the sub-set of features supported by FVID2. Please refer to the individual driver userGuide for the features exposed by drivers.

FVID2 enumerations

FVID2 Data Format

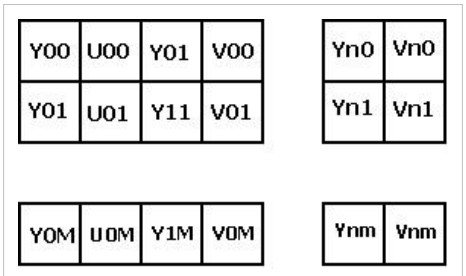
`Fvid2_DataFormat` represents the arrangement of the different components forming the pixel. These components can be in YUV color space or the RGB color space or any other color space. Below figure shows the commonly used data formats. FVID2 supports many more data formats. Specific driver may expose subset of the data formats from the mentioned below based on the hardware capability.

YUV420 Semiplanar Format



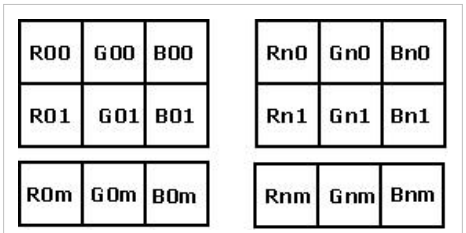
YUV420 DataFormat with Y in one plane and UV in another plane

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 CbParams

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              chNum;
    /**< 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

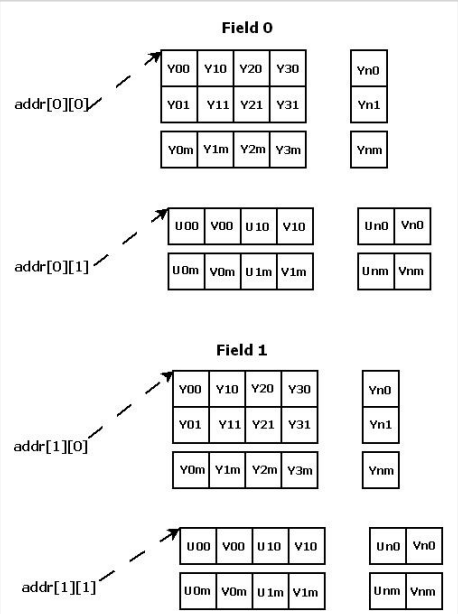
```
typedef struct
{
    UInt32              sliceNum;
        /**< Current slice Number in this frame,
             range is from 0 to (NoOfSlicesInFrame-1)  */
    UInt32              numSlcInLines;
        /**< Number of lines available in the frame at the end of
this slice. */
    UInt32              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

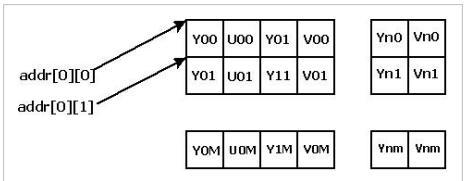
Below figure shows the `addr` field of the Fvid2_Frame structure for the YUV420 semi planar data in which the image is interlaced and fields are separate in two different buffers.



`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;
```

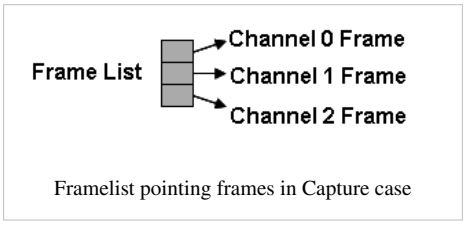
FVID2 FrameList

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.

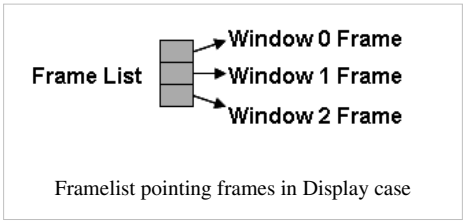
```
typedef struct
{
    Fvid2_Frame        *frames[FVID2_MAX_FVID_FRAME_PTR];
        /**< An array of FVID2 frame pointers.
             [IN] The content of the pointer array i.e Fvid2_Frame
pointer is input
             for queue operation
             [OUT] Output for dequeue operation. */
    UInt32              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

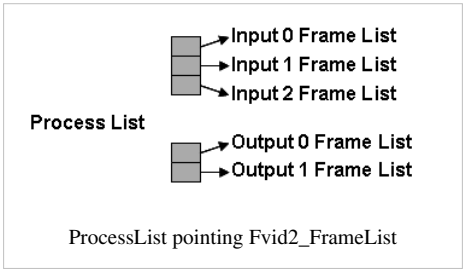
FVID2 ProcessList

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.

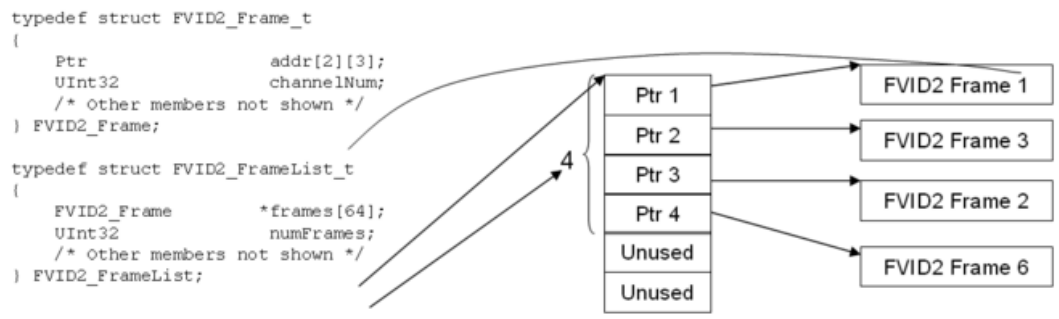
```
Int32 Fvid2_dequeue(Fvid2_Handle handle,
                    Fvid2_FrameList *frameList,
                    UInt32 streamId,
                    UInt32 timeout);
```

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.

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.

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.

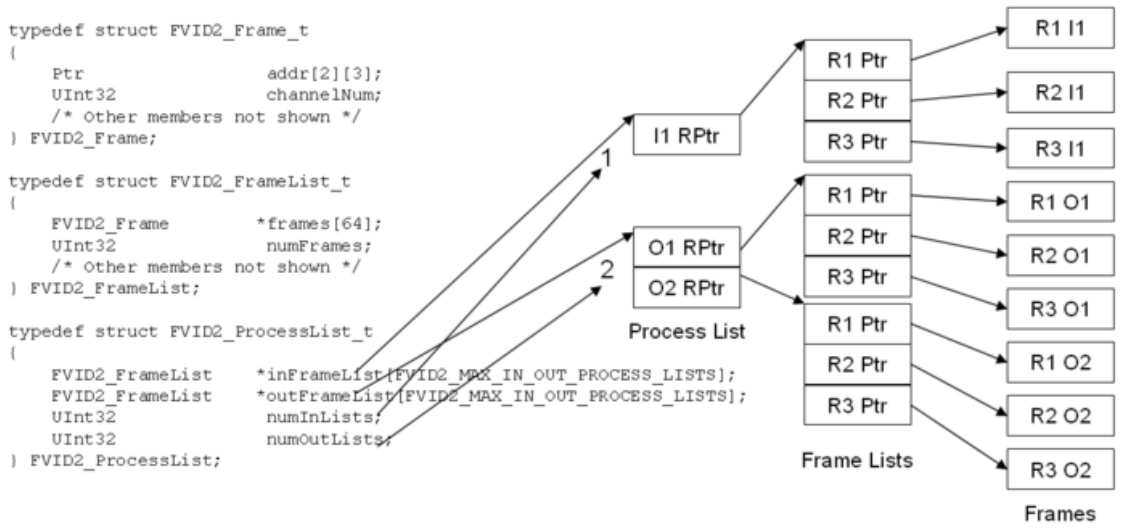
FVID2 GetProcessedFrames

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

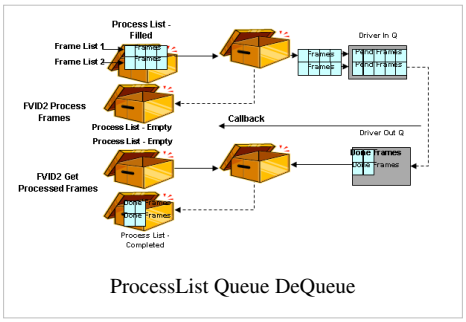


FVID2 ProcessList

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 Get Standard Info

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. */
```

UserGuideBSPPlatformAPIs

Platform APIs and Drivers

Introduction

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

Common Init

This is a common initialization routine that initializes the BSS and VPDMA descriptor memory address and size.

Board Init

This is the application board initialization function. For every new application board support to be added the user shall be required to add the board-specific file which shall comprise of the board-detection mechanism, the revision details.

```
Int32 Bsp_boardInit(const Bsp_BoardInitParams *initPrms)
```

initPrms - Board initialization parameters. Its explained below. **return value** - Returns `BSP_SOK` on success, else proper error code.

```
/**
 *  \brief Board initialization parameters.
 */
typedef struct
{
    Bsp_BoardId  boardId;
    /**< Override board ID detection. Set this to #BSP_BOARD_MAX to
allow
     *   auto detection of the board connected. Setting any other value
 will
     *   override the auto detect and force the board ID to the
supplied
     *   value. */
    Bsp_BoardRev baseBoardRev;
    /**< Override base board revision detection. Set this to
#BSP_BOARD_REV_MAX
     *   to allow auto detection of base board revision. Setting any
other
     *   value will override the auto detect and force the revision to
the
     *   supplied value. */
    Bsp_BoardRev dcBoardRev;
    /**< Override daughter card board revision detection. Set this
to
```

```
     *   #BSP_BOARD_REV_MAX to allow auto detection of DC board
revision.
     *   Setting any other value will override the auto detect and
force
     *   the revision to the supplied value. */
} Bsp_BoardInitParams;
```

Platform Init

This is the platform initialization functions. It sets up the hardware for BSP video drivers. This function is platform dependent and it needs to be ported for different platforms like TDA2SEDX, TI814x, etc. Function does following at at a high level for setting up of platform.

- Enabling of the VPS functional clocks.
- Setting up of the pin mux for VIP capture for 24/16 bit data signals and 5 control signals.
- Setting up of the interrupt muxing if required.

```
Int32 Bsp_platformTI814xInit(const Bsp_PlatformInitParams *initParams)
```

initParams - Platform initialization parameters. Its explained below. **return value** - Returns `BSP_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 processor. */
} Bsp_PlatformInitParams;
```

Fvid2_init

This function 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.

```
Int32 Fvid2_init(Ptr args)
```

args - User should always pass `NULL` here.

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

VPS Init

This function shall initialize the Video Processing System data structures for the underlying HAL layer, CORE layer.

The input argument to this function is const Vps_InitParams *initPrms. This initParams comprises of:

- Parameters to enable address translation of descriptor memory (virtToPhys)
- Parameters to enable cache operation if descriptors are present in cacheable section

```
/**
 *  \brief VPS initialization parameters.
 */
typedef struct
{
    Bool isAddrTransReq;
    /**< Set this flag to TRUE if the driver has to perform address
translation
     *   of the descriptor memory before submitting the descriptor to
the
     *   hardware. This is used when the physical memory of the
descriptor
     *   is mapped to a different virtual memory.
     *
     *   When address translation is enabled, the dirver performs the
following
     *   operations to convert the virtual address to physical address
and
     *   vice versa.
     *
     *   physAddr = (virtAddr - virtBaseAddr) + physBaseAddr;
     *   virtAddr = (physAddr - physBaseAddr) + virtBaseAddr;
     *
     *   Important: The descriptor memory should in a physically
continuous
     *   memory.
     *
     *   Note: The buffer address will not be translated using the
above
     *   translation and hence the application should provide the
physical
     *   address to be programmed to the hardware.
     *
     *   Note: VPSHAL_VPDMA_ENABLE_ADDR_TRANS macro should be defined
in
     *   vpshal_vpdma.c file to enable address translation at compile
time.
     *   By default this is defined. But alternatively application
could disable
     *   this conversion at compile time by removing this macro
definition
```

```
     *   to improve performance. */
    UInt32 virtBaseAddr;
    /**< Virtual memory base address. */
    UInt32 physBaseAddr;
    /**< Physical memory base address. */
    Bool   isCacheOpsReq;
    /**< This will enable cache flush and invalidate operations on
the
     *   descriptor memory in case the descriptor memory is cache
region.
     *
     *   Note: This is not supported in the current implementation and
is meant
     *   for future use. */
} Vps_InitParams;
```

I2C Init

This function shall initialize BSP I2C driver. This routine takes in two arguments:

- Number of I2C instances to be initialized
- Bsp_I2cInitParams i2cInitParams details of which are given below

```
/**
 *  \brief I2C Instance Configuration Object
 *
 *  This structure provides various configuration options for I2C
controller.
 *  It needs to be passed while creating the specific I2C instance
object.
 */
typedef struct
{
    UInt32 instId;
    /**< I2C controller number. */

    UInt32 opMode;
    /**< Driver operating mode - polled or interrupt. Currently only
 polled
     *   mode is supported.
     *   For valid values see #Bsp_I2cOpMode. */
    UInt32 isMasterMode;
    /**< Operating in Master/Slave mode:
     *   0: Slave mode
     *   1: Master mode
     *   Currently only master mode is supported. */
    UInt32 is10BitAddr;
    /**< Addressing mode:
     *   0: 7 bit addressing mode.
```

```
     *   1: 10 bit addressing mode. */
    UInt32 i2cBusFreq;
    /**< I2C Bus Frequency (in KHz). */
    UInt32 i2cOwnAddr;
    /**< Own address (7 or 10 bits). */
    Ptr    i2cRegs;
    /**< I2C peripheral base address. */
    UInt32 i2cIntNum;
    /**< Interrupt number. */
} Bsp_I2cInitParams;
```

Device Init

This function shall initialize all the devices such as video decoders TVP7002, TVP5158, sensors MT9V022, etc. For every new application board support to be added the user shall be required to add the board-specific device file which shall comprise of the device initialization requirements.

```
Int32 Bsp_deviceInit(const Bsp_DeviceInitParams *pPrm)
```

pPrm - Device initialization parameters. Its explained below. **return value** - Returns `BSP_SOK` on success, else proper error code.

```
/**
 *  \brief External video device sub-system init parameters
 */
typedef struct
{
    UInt32 isI2cInitReq;
    /**< Indicates whether I2C initialization is required. */
    UInt32 isI2cProbingReq;
    /**< If this is TRUE, THE INIT will try to probe all the I2C
     *   devices connected on a specific I2C bus. This should be FALSE
for
     *   all production system/build 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 is FALSE. */
} Bsp_DeviceInitParams;
```

Device DeInit

This function shall de-initialize all the devices such as video decoders TVP7002, TVP5158, sensors MT9V022, etc. For every new application board support to be added the user shall be required to add the board-specific device file which shall comprise of the device de-initialization requirements.

```
Int32 Bsp_deviceDeInit(Ptr args)
```

args - User should always pass `NULL` here.

return value - Returns `BSP_SOK` on success, else proper error code.

I2C DeInit

This function shall de-initialize the I2C driver.

```
Int32 Bsp_i2cDeinit(Ptr args)
```

args - User should always pass `NULL` here.

return value - Returns `BSP_SOK` on success, else proper error code.

VPS DeInit

This function shall de-initialize the HAL and Core elements.

```
Int32 Vps_deInit(Ptr args)
```

args - User should always pass `NULL` here.

return value - Returns `BSP_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.

Platform DeInit

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 Bsp_platformDeInit(Ptr args)
```

args - User should always pass `NULL` here.

return value - Returns `BSP_SOK` on success, else proper error code.

This inturn shall call a SoC specific de-initialization routine.

```
Int32 Bsp_platformTI814xDeInit(void)
```

return value - Returns `BSP_SOK` on success, else proper error code.

Board DeInit

This is application board de-Initialization function.

```
Int32 Bsp_boardDeInit(Ptr args)
```

args - User should always pass `NULL` here.

return value - Returns `BSP_SOK` on success, else proper error code.

Common DeInit

This is common de-Initialization function.

```
Int32 Bsp_commonDeInit(Ptr args)
```

args - User should always pass `NULL` here.

return value - Returns `BSP_SOK` on success, else proper error code.

This internally shall free the BSS and VPDMA descriptor memory space.

UserGuideBSPCaptureDriver

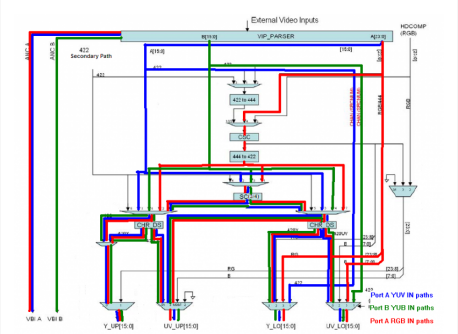
Introduction

VIP capture driver makes use of VIP hardware block in Tda2xx to capture data from external video source like video decoders (example, TVP5158, SIL9127) or sensors (example OV1063X). 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 or an OCMC buffer.

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

Important

The 422 secondary path and HDCOMP (RGB) path as shown in the figure below are wired out for Tda2xx.



Vip-capture-paths.png

Features supported

Features	**Supported in Tda2xx**	**Supported in TI814x**
Input Video Source Formats		
YUV422 8-bit embedded sync mode	YES	YES
YUV422 16-bit embedded sync mode	NOT TESTED	YES
RGB 24-bit embedded sync mode	NOT TESTED	NOT TESTED
YUV422 8-bit discrete sync mode	YES	NOT TESTED
YUV422 16-bit discrete sync mode	YES	YES
RGB 24-bit discrete sync mode	NOT TESTED	NOT TESTED
YUV422 8-bit 2x/4x pixel multiplxed mode	NO	NO
YUV422 8-bit 4x line multiplexed mode	NO	NO
YUV422 8-bit 4x split-line multiplexed mode	NO	NO
YUV444 24-bit embedded/discrete sync mode	NOT TESTED	NOT TESTED
Output Video formats		
YUV422 YUYV interleaved format	YES	YES
YUV420 Semi-planer format	YES	YES
RGB 24-bit interleaved format	YES	YES
YUV422 Semi-planer format	YES	NO
In-line video processing features		
Color space conversion	YES	YES
Down-Scaling	YES	YES
Chroma-down sampling	YES	YES
Other features		
Multi-instance (VIP1, VIP2, VIP3), multi-slice (S0, S1), multi-port capture (Port A, Port B), with ability to configure each instance, port independently	YES	YES
Interlaced as well as progressive capture	YES	YES
Per frame info to user like - field ID, captured frame width x height, timestamp, logical channel ID	YES	YES
Frame-rate depends on external video source, no limitation in driver as such	YES	YES
For RGB input, optional color space conversion to YUV is supported	NOT TESTED	NOT TESTED
For RGB input with color space conversion to YUV enabled, optional scaling is supported	NOT TESTED	NOT TESTED
For YUV input, optional color space conversion to RGB is supported	YES	NOT TESTED
For YUV input, optional scaling is supported	YES	YES
For YUV input, optional chroma downsampling is supported	YES	YES
Per channel frame-dropping. Example, for a 60fps video source, 30fps, 15fps, 7fps capture	YES	YES
Ability to change scalar parameters while capture is running	NO	NO
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	NO
Raw VBI capture for single/multi channel modes	NO	NO

Non-blocking FVID2 queue, dequeue API support	YES	YES
Possible to configure VIP port for different video input source properties like Hsync polarity, Vsync polarity, PCLK polarity	YES	YES
Tiler memory support when output type is YUV420 semi-planer	NOT TESTED	NOT TESTED
Sub-frame based capture	YES	YES

- 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	Supoprt in Tda2xx
YUV422 8/16-bit embedded/discrete sync mode	YUV422 YUYV interleaved format (optionally scaled)	NONE	YES
	YUV420 Semi-planer format (optionally scaled)	NONE	YES
	RGB 24-bit interleaved format (via CSC)	NONE	YES
	YUV422 Semi-planer format (optionally scaled)	NONE	YES
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	YES
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format (via CSC)	YES
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	YES
	YUV422 Semi-planer format (one output optionally scaled)	YUV422 YUYV interleaved format (one output optionally scaled)	YES
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	YES
	YUV420 Semi-planer format (optionally scaled)	RGB 24-bit interleaved format (via CSC)	YES
YUV444 24-bit embedded/discrete sync mode	NA	NA	NO

RGB 24-bit discrete sync mode (CSC is used when output format is YUV)	YUV422 YUYV interleaved format (optionally scaled)	NONE	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	NONE	NOT TESTED
	RGB 24-bit interleaved format	NONE	NOT TESTED
	YUV422 Semi-planer format (optionally scaled)	NONE	NO
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	NOT TESTED
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NO
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	NOT TESTED
	RGB 24-bit interleaved format	YUV420 Semi-planer format (optionally scaled)	NOT TESTED
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NO
RGB 16-bit discrete sync mode (CSC is used when output format is YUV)	YUV422 YUYV interleaved format (optionally scaled)	NONE	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	NONE	NOT TESTED
	RGB 24-bit interleaved format	NONE	NOT TESTED
	YUV422 Semi-planer format (optionally scaled)	NONE	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	NOT TESTED
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NOT TESTED
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	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
	YUV420 Semi-planer format (optionally scaled)	NONE	NOT TESTED
	RGB 24-bit interleaved format	NONE	NOT TESTED
	YUV422 Semi-planer format (optionally scaled)	NONE	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	NOT TESTED
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NOT TESTED
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	NOT TESTED
	YUV420 Semi-planer format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	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
	YUV420 Semi-planer format (optionally scaled)	NONE	NOT TESTED
	RGB 24-bit interleaved format	NONE	NOT TESTED
	YUV422 Semi-planer format (optionally scaled)	NONE	NO
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV420 Semi-planer format (one output optionally scaled)	NOT TESTED
	YUV422 YUYV interleaved format (optionally scaled)	RGB 24-bit interleaved format	NOT TESTED
	YUV422 YUYV interleaved format (one output MUST BE scaled)	YUV422 YUYV interleaved format (one output MUST be scaled)	NOT TESTED
	YUV422 YUYV interleaved format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	NO
	YUV420 Semi-planer format (one output MUST be scaled)	YUV420 Semi-planer format (one output MUST be scaled)	NOT TESTED
	RGB 24-bit interleaved format	YUV420 Semi-planer format (optionally scaled)	NOT TESTED
	YUV420 Semi-planer format (one output optionally scaled)	YUV422 Semi-planer format (one output optionally scaled)	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
- Control Phase: Control call to set the capture parameters. Here the driver, core and hal is initialized with the instance related setup parameters.
- 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 BSP system init. Below code shows the FVID2 API used to initialize the overall Capture 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. In the below example, prior to the global VIP capture handle creation, BspUtils_appDefaultInit() routine is called. This function initializes the board, platform, fvid2, i2c, device drivers.

```
static void CaptApp_init(CaptApp_Obj *appObj)
{
    Int32  retVal;
    UInt32 isI2cInitReq;

    /* System init */
    isI2cInitReq = TRUE;
    retVal       = BspUtils_appDefaultInit(isI2cInitReq);
    if (retVal != FVID2_SOK)
    {
        GT_0trace(BspAppTrace, GT_ERR,
                  APP_NAME ": System Init Failed!!!\n");
        return;
    }
```

```
    /* Create global capture handle, used for common driver
configuration */
    appObj->fvidHandleAll = Fvid2_create(
        FVID2_VPS_CAPT_VID_DRV,
        VPS_CAPT_INST_ALL,
        NULL,                           /* NULL for VPS_CAPT_INST_ALL
*/
        NULL,                           /* NULL for VPS_CAPT_INST_ALL
*/
        NULL);                          /* NULL for VPS_CAPT_INST_ALL
*/
    if (NULL == appObj->fvidHandleAll)
    {
        GT_0trace(BspAppTrace, GT_ERR,
                  APP_NAME ": Global Handle Create Failed!!!\n");
        return;
    }

    GT_0trace(BspAppTrace, GT_INFO,
              APP_NAME ": CaptApp_init() - DONE !!!\n");

    return;
}
```

Create Phase

In this phase user application opens or creates a driver instance. A driver instance is associated with one or more VIP instances, slices and parser ports depending on whether the operation mode is 8-bit or 16-bit or 24-bit. Based on the VIP instance, slice and port information user can use the below routine to compute the instanceId to be passed to the Fvid2_Create() routine.

```
/**
 *  \brief Macro to generate VIP capture driver instance ID to be passed
 *  during create parameter.
 *
 *  vipId   -   VPS_VIP1, VPS_VIP2 or VPS_VIP3<br>
 *  sliceId -   VPS_VIP_S0 or VPS_VIP_S1<br>
 *  portId  -   VPS_VIP_PORTA or VPS_VIP_PORTB
 */
#define VPS_CAPT_VIP_MAKE_INST_ID(vipId, sliceId, portId) \
    ((portId) +                                           \
     ((sliceId) * (VPS_VIP_PORT_MAX)) +                   \
     ((vipId) * (VPS_VIP_SLICE_MAX * VPS_VIP_PORT_MAX)))
```

Control Phase

In this phase, user application provides the detailed set of capture VIP parameters to the driver.

```
retVal = Fvid2_control(
     instObj->drvHandle,
     IOCTL_VPS_CAPT_SET_VIP_PARAMS,
     &instObj->vipPrms,
     NULL);
```

Driver Instance to hardware port mapping for different bus-widths

The mapping of driver instance to VIP parser ports in HDVPSS is shown below:

VIP Instance	Slice Id	Port Id	8-bit interface	16-bit interface	24-bit interface
VIP 1	Slice 0	Port A	VIP1 S0 PortA	VIP1 S0 PortA	VIP1 S0 PortA
VIP 1	Slice 0	Port B	VIP1 S0 PortB	NOT USED	NOT USED
VIP 1	Slice 1	Port A	VIP1 S1 PortA	VIP1 S1 PortA	VIP1 S1 PortA
VIP 1	Slice 1	Port B	VIP1 S1 PortB	NOT USED	NOT USED
VIP 2	Slice 0	Port A	VIP2 S0 PortA	VIP2 S0 PortA	VIP2 S0 PortA
VIP 2	Slice 0	Port B	VIP2 S0 PortB	NOT USED	NOT USED
VIP 2	Slice 1	Port A	VIP2 S1 PortA	VIP2 S1 PortA	VIP2 S1 PortA
VIP 2	Slice 1	Port B	VIP2 S1 PortB	NOT USED	NOT USED
VIP 3	Slice 0	Port A	VIP3 S0 PortA	VIP3 S0 PortA	VIP3 S0 PortA
VIP 3	Slice 0	Port B	VIP3 S0 PortB	NOT USED	NOT USED
VIP 3	Slice 1	Port A	VIP3 S1 PortA	VIP3 S1 PortA	VIP3 S1 PortA
VIP 3	Slice 1	Port B	VIP3 S1 PortB	NOT USED	NOT USED

Output streams

A maximum of four output streams (including 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). Currently multi-channel or muxed mode capture is not supported by the VIP driver.

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_CaptVipParams).

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.

```
    /* Start driver */
    for (instCnt = 0u; instCnt < appObj->testPrms.numHandles;
instCnt++)
    {
        instObj = &appObj->instObj[instCnt];

        retVal = Fvid2_start(instObj->drvHandle, NULL);
        if (retVal != FVID2_SOK)
        {
            GT_0trace(BspAppTrace, GT_ERR,
                      APP_NAME ": Capture Start Failed!!!\n");
            return;
        }
    }
```

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.

```
    /* Stop driver */
    for (instCnt = 0u; instCnt < appObj->testPrms.numHandles;
instCnt++)
    {
        instObj = &appObj->instObj[instCnt];
        retVal  = Fvid2_stop(instObj->drvHandle, NULL);
        if (retVal != FVID2_SOK)
        {
            GT_0trace(BspAppTrace, GT_ERR,
                      APP_NAME ": Capture Stop Failed!!!\n");
            return;
        }
    }
```

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 "chNum" 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->chNum),        // VIP instance ID
    Vps_captGetStreamId(pCurFrame->chNum),// Stream ID
    Vps_captGetChId(pCurFrame->chNum),    // 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.chNum

Important

Be careful to not modify the "Fvid2_Frame.chNum" from the received (dequeued) frame when returning (queuing) the frame back to the driver. The FVID2 queue API uses "chNum" 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.chNum" identifies the channel associated with a given frame. Given "Fvid2_Frame.chNum" 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

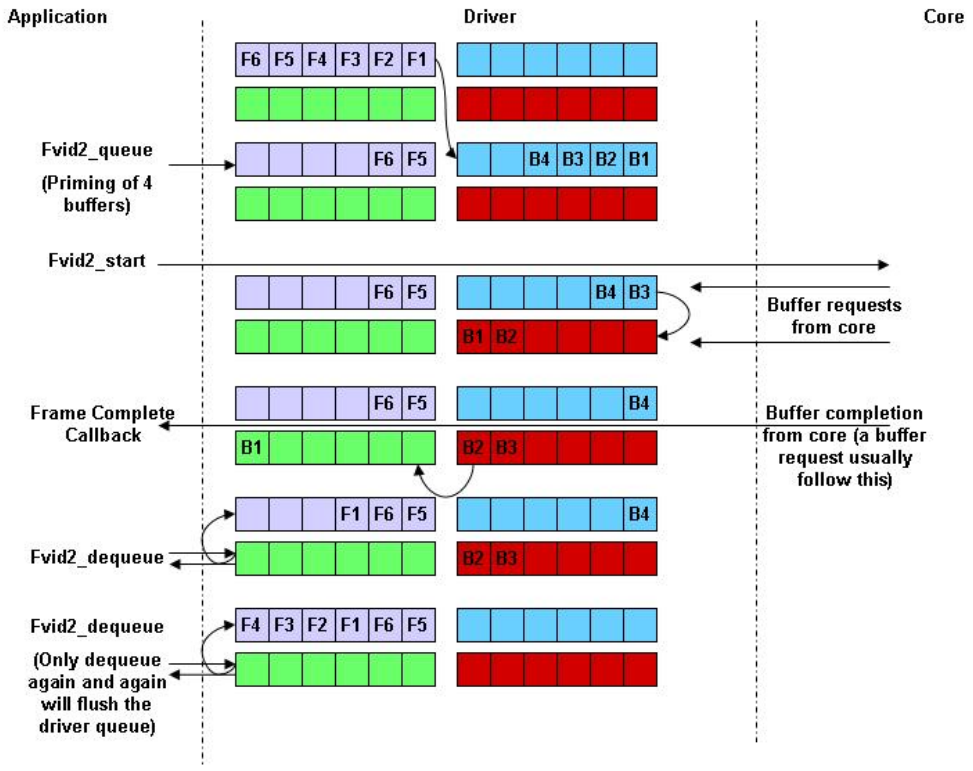
VIP Instance	Output Stream 0	Output Stream 1	Output Stream 2	Output Stream 3
VIP0 Port A	CH00 .. CH15	CH16 .. CH31	CH32 .. CH47	CH48 .. CH63
VIP0 Port B	CH64 .. CH79	CH80 .. CH95	CH96 .. CH111	CH112 .. CH127
VIP1 Port A	CH128 .. CH143	CH144 .. CH159	CH160 .. CH175	CH176 .. CH191
VIP1 Port B	CH192 .. CH207	CH208 .. CH223	CH224 .. CH239	CH240 .. CH255

Buffer Capture Mode (BCM)

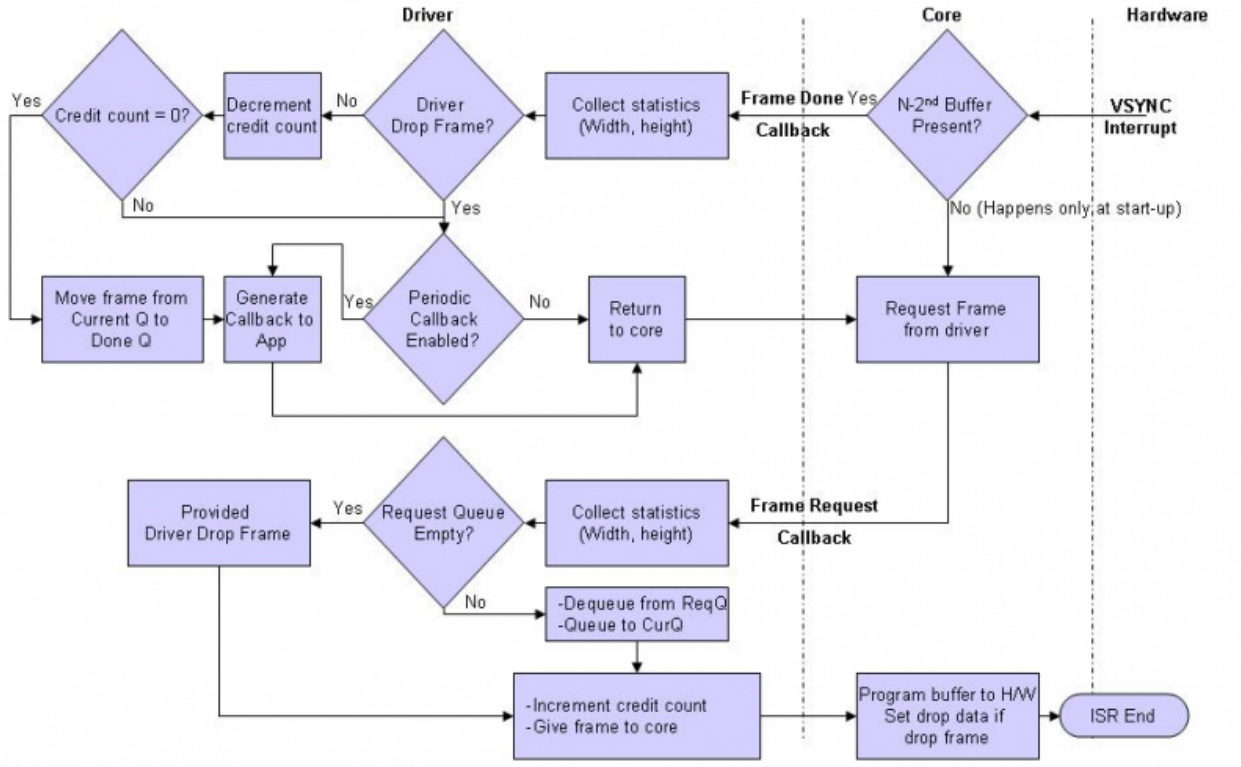
The section explains the design of the capture driver for various buffer capture mode. The decision of queuing and de-queuing a buffer to the core happens at the frame completion callback from the core which in turn is equivalent to VSYNC of the capture hardware.

- **Frame drop mode**

In this mode the driver will stop capturing data when there are no more buffers at the input queue. The driver will not hold any buffer with it and the last buffer will be returned to the application through de-queue call. For this mode, the driver makes use of the VPDMA drop data feature.



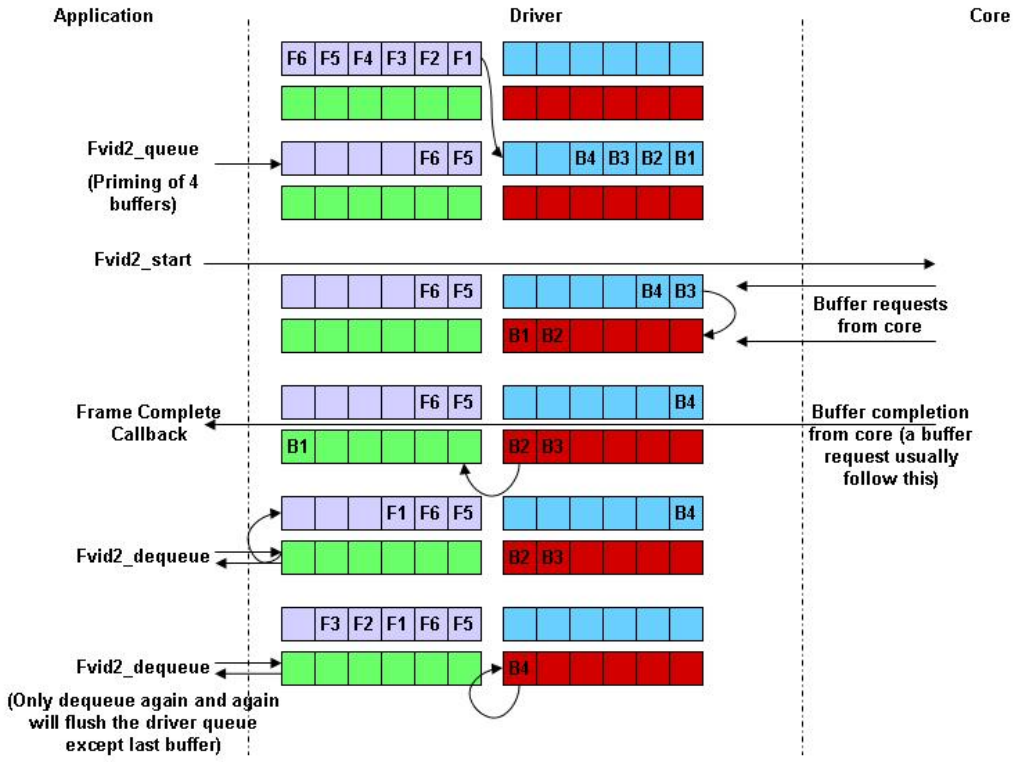
Frame Drop Mode Buffer Flow



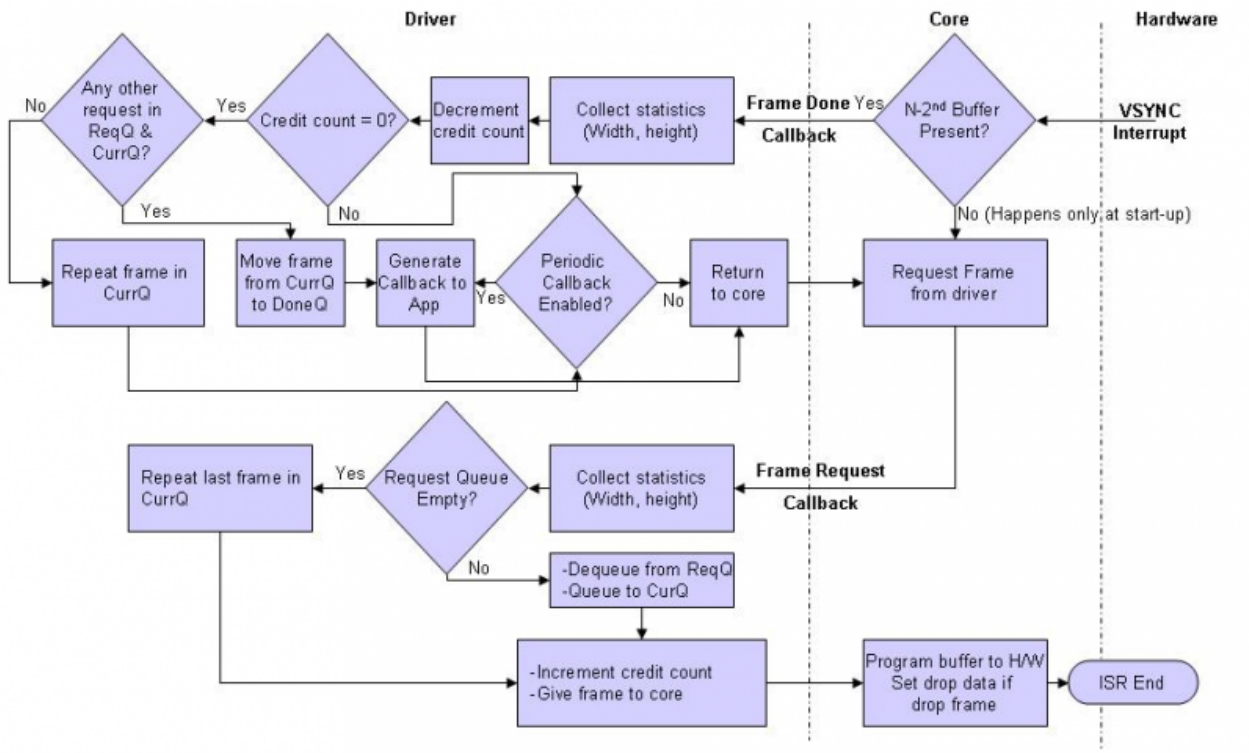
Frame Drop Mode ISR Flow Diagram

- **Last frame repeat mode**

In this mode the driver will keep capturing the data to the last queued buffer when there are no more buffers at the input queue. The driver will hold the last buffer with it till the application queues any new buffer or the capture is stopped



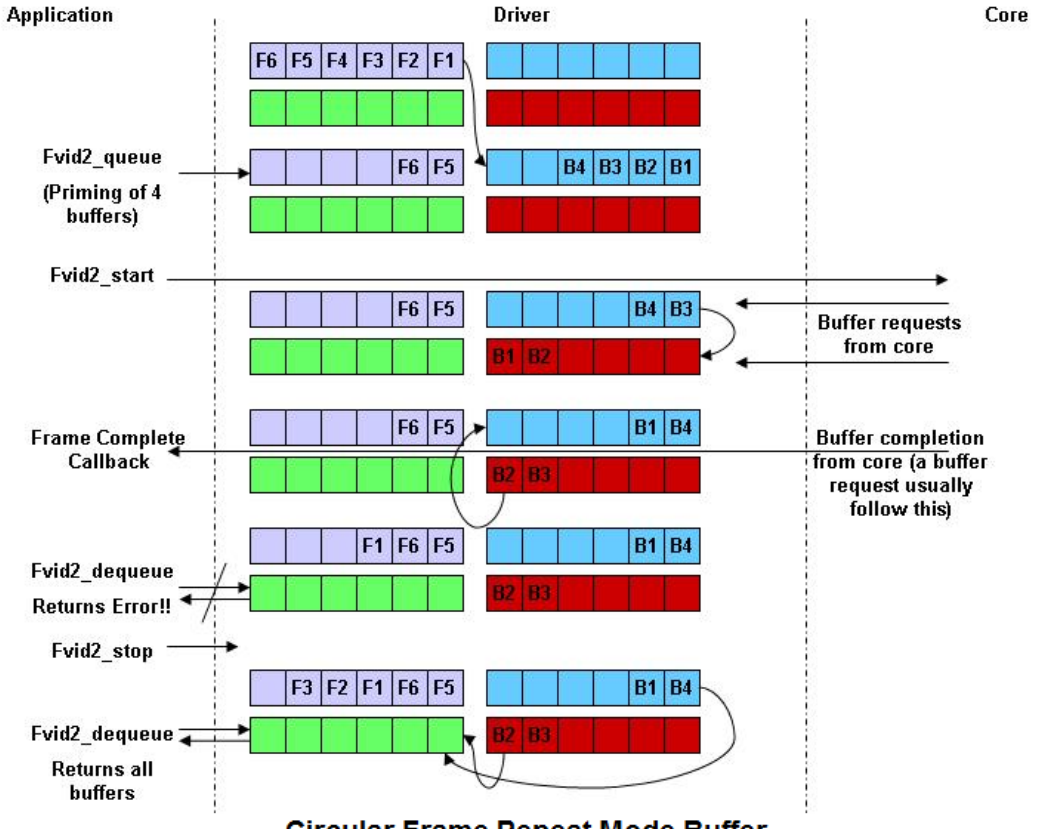
Last Frame Repeat Mode Buffer Flow



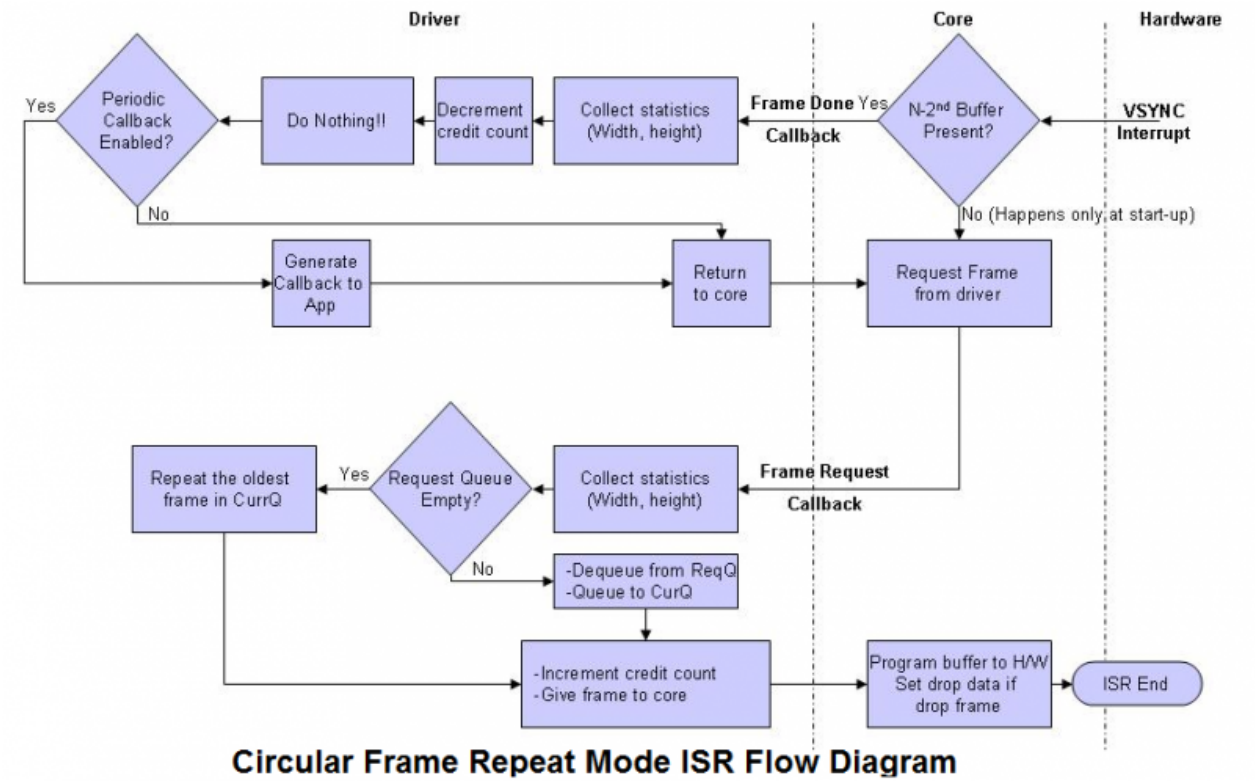
Last Frame Repeat Mode ISR Flow Diagram

- **Circular frame repeat mode**

In this mode the driver will keep reusing all the sets of buffer with it in a circular fashion. Application cannot get back any buffer from the driver when streaming is on and dequeue call will result in error.



Circular Frame Repeat Mode Buffer



Circular Frame Repeat Mode ISR Flow Diagram

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;

// chNum is the one that was specified by user during create in
chNumMap[][]
frameSkip.chNum = createArgs.chNumMap[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
          );
```

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.chNum = createArgs.chNumMap[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
                );
    }
  }
}
```

Max Size IOCTL_VPS_CAPT_SET_VIP_MAX_SIZE

User can program the three MAX_SIZE registers (MAX_SIZE_REG1, MAX_SIZE_REG2, MAX_SIZE_REG3) for each VPDMA associated with VIP1, VIP2, and VIP3 using this IOCTL. This IOCTL is valid only for Tda2xx. The Centaurus hardware does not have these registers present. The user application can program these registers using the global VIP capture handle before even creating the driver handle for the actual capture instance.

An example is given below:

```
/**
 * There are 3 32-bit MAX_SIZE registers supported for Tda2xx platform
family.
 * These registers provide two parameters width[31:16] and
height[15:0].
 * The VPDMA transmits to external buffer the maximum out width number
of
 * pixels and maximum out height number of pixel lines.
 * If the VIP receives data exceeding the maximum out width/height then
 it
 * continues to capture the data. VPDMA will not transfer it to the
 * external buffer.
 * This register (if used) should have valid range of values.
 * The valid range for maximum out width shall be [1, 4096]
 * The valid range for maximum out height shall be [1, 2048]
 * Example: For a YUV420SP capture,
 * For luma, the maximum out [width, height] can go up to [2048, 2048].
 * For chroma, the maximum out [width, height] can go up to [2048,
1024].
 * Example: For a YUV422I capture,
 * For luma, the maximum out [width, height] can go up to [4096, 2048].
 */
/* MAX SIZE Register Width and Height configurations */
#define CAPT_APP_MAXSIZE_1_WIDTH        (1920u)
#define CAPT_APP_MAXSIZE_2_WIDTH        (1920u)
#define CAPT_APP_MAXSIZE_3_WIDTH        (1280u)
#define CAPT_APP_MAXSIZE_1_HEIGHT       (1080u)
#define CAPT_APP_MAXSIZE_2_HEIGHT       (540u)
#define CAPT_APP_MAXSIZE_3_HEIGHT       (800u)

        instObj->maxOutWidth[0u]  = CAPT_APP_MAXSIZE_1_WIDTH;
        instObj->maxOutHeight[0u] = CAPT_APP_MAXSIZE_1_HEIGHT;
        instObj->maxOutWidth[1u]  = CAPT_APP_MAXSIZE_2_WIDTH;
```

```
        instObj->maxOutHeight[1u] = CAPT_APP_MAXSIZE_2_HEIGHT;
        instObj->maxOutWidth[2u]  = CAPT_APP_MAXSIZE_3_WIDTH;
        instObj->maxOutHeight[2u] = CAPT_APP_MAXSIZE_3_HEIGHT;

        if (Bsp_platformIsTda2xxFamilyBuild())
        {
            VpsVpdmaMaxSizeParams_init(&vipMaxSizePrms);
            vipMaxSizePrms.instId =
                Vps_captGetVipId(appObj->testPrms.instId[instCnt]);
            vipMaxSizePrms.maxOutWidth[0u]  =
instObj->maxOutWidth[0u];
            vipMaxSizePrms.maxOutHeight[0u] =
instObj->maxOutHeight[0u];
            vipMaxSizePrms.maxOutWidth[1u]  =
instObj->maxOutWidth[1u];
            vipMaxSizePrms.maxOutHeight[1u] =
instObj->maxOutHeight[1u];
            vipMaxSizePrms.maxOutWidth[2u]  =
instObj->maxOutWidth[2u];
            vipMaxSizePrms.maxOutHeight[2u] =
instObj->maxOutHeight[2u];

            retVal = Fvid2_control(
                appObj->fvidHandleAll,
                IOCTL_VPS_CAPT_SET_VIP_MAX_SIZE,
                &vipMaxSizePrms,
                NULL);
            if (retVal != FVID2_SOK)
            {
                GT_0trace(
                    BspAppTrace, GT_ERR,
                    APP_NAME
                    ": VIP Set Max Frame Size Params IOCTL
Failed!!!\n");
                return;
            }
        }
```

Set VIP Parameters IOCTL_VPS_CAPT_SET_VIP_PARAMS

The application user after creating the instance handle shall need to make a control call with VIP specific parameters to configure the VIP blocks.

An example is given below:

```
        retVal = Fvid2_control(
            instObj->drvHandle,
            IOCTL_VPS_CAPT_SET_VIP_PARAMS,
            &instObj->vipPrms,
            NULL);
```

```
        if (retVal != FVID2_SOK)
        {
            GT_0trace(BspAppTrace, GT_ERR,
                      APP_NAME ": VIP Set Params IOCTL Failed!!!\n");
            return;
        }
```

Get VIP Parameters IOCTL_VPS_CAPT_GET_VIP_PARAMS

The application user after creating the instance handle shall need to make a control call to get the default VIP parameters.

An example is given below:

```
        retVal = Fvid2_control(
            instObj->drvHandle,
            IOCTL_VPS_CAPT_GET_VIP_PARAMS,
            &instObj->vipPrms,
            NULL);
        if (retVal != FVID2_SOK)
        {
            GT_0trace(BspAppTrace, GT_ERR,
                      APP_NAME ": VIP Get Params IOCTL Failed!!!\n");
            return;
        }
```

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:

```
        retVal = Fvid2_delete(instObj->drvHandle, NULL);
        if (FVID2_SOK != retVal)
        {
            GT_0trace(BspAppTrace, GT_ERR,
                      APP_NAME ": Capture Delete Failed!!!\n");
            return;
        }
```

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

```
static void CaptApp_deInit(CaptApp_Obj *appObj)
{
    Int32  retVal;
    UInt32 isI2cDeInitReq;

    /* Delete global VIP capture handle */
    retVal = Fvid2_delete(appObj->fvidHandleAll, NULL);
    if (retVal != FVID2_SOK)
    {
        GT_0trace(BspAppTrace, GT_ERR,
                  APP_NAME ": Global handle delete failed!!!\n");
        return;
    }

    /* System de-init */
    isI2cDeInitReq = TRUE;
    retVal         = BspUtils_appDefaultDeInit(isI2cDeInitReq);
    if (retVal != FVID2_SOK)
    {
        GT_0trace(BspAppTrace, GT_ERR,
                  APP_NAME ": System De-Init Failed!!!\n");
        return;
    }

    GT_0trace(BspAppTrace, GT_INFO,
              APP_NAME ": CaptApp_deInit() - DONE !!!\n");
}
```

UserGuideBSPM2mVpeDriver

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 same/different frame parameters like height, width etc.
- Memory driver supports parameter configuration for the buffer processing 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.

VPE Memory to Memory Driver

Introduction

This chapter describes the hardware overview, application software interfaces, typical application flow and sample application usage for VPE 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 BSP driver release to another. See respective release notes for exact release specific details.

Features Supported

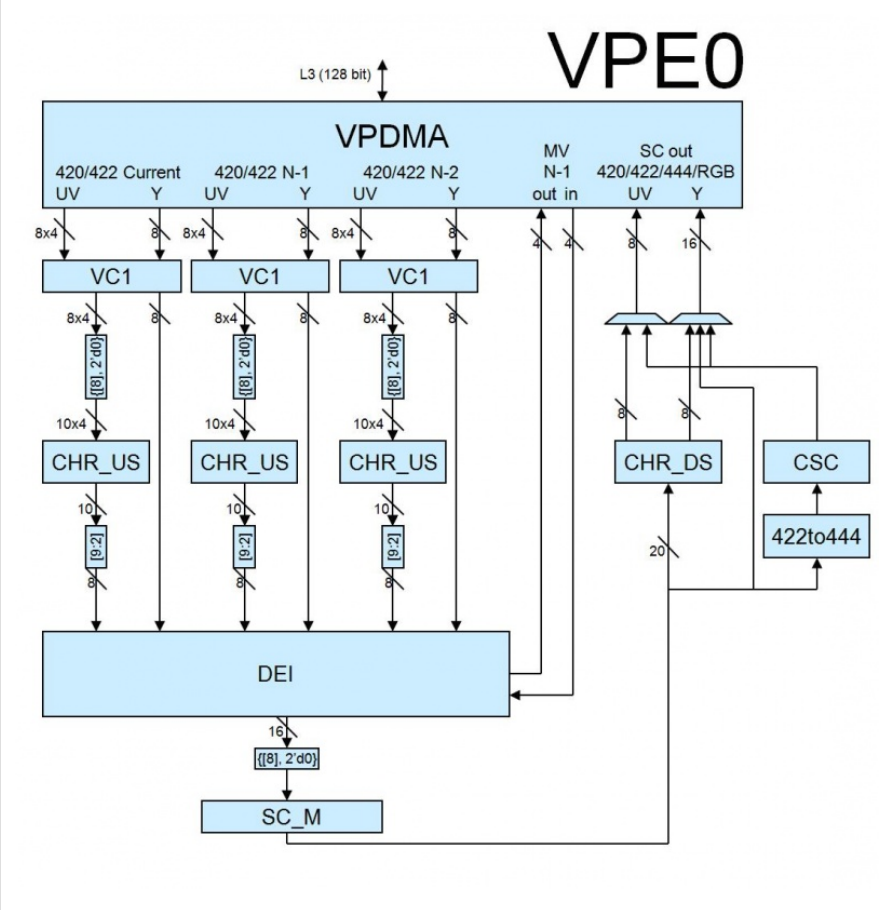
Features	Supported in TDA2SEDx	Supported in TI814x
Instances		
VPS_M2M_INST_VPE1 single scale path	YES	YES
Input Formats		
YUV422 Interleaved	YES	YES
YUV420 Semi-Planar	YES	YES
YUV422 Semi-Planar	YES	YES
YUV422 Semi-Planar Tiled	NOT TESTED	NOT TESTED
YUV420 Semi-Planar Tiled	NOT TESTED	NOT TESTED
Output Formats		
YUV422 Interleaved	YES	YES
YUV420 Semi-Planar	YES	NO/NA
YUV422 Semi-Planar	YES	NO/NA
YUV420 Semi-Planar Tiled	NOT TESTED	NOT TESTED
YUV422 Semi-Planar Tiled	NOT TESTED	NOT TESTED
RGB888	YES	NO/NA
YUV444	YES	NO/NA
DEI Features		
DEI in deinterlacing mode	YES	YES
DEI in progressive bypass mode	YES	YES
Line averaging and field averaging mode of DEI operation	YES	YES
SC Features		
Optional scaling using SC1	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
Lazy loading of coefficient	YES	YES
User programmable scalar coefficients	NO	NO
Other Features		
Frame drop feature to enable load balancing	NOT TESTED	NOT TESTED
Multi-channel (up to `VPS_M2M_MAX_CH_PER_INST` channels per instance)	YES	YES
Multi-handle (up to `VPS_M2M_MAX_CH_PER_HANDLE` 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	YES

Input resolution change when DEI is in deinterlacing mode	YES	YES
Output resolution change	YES	YES
SC crop and config change on SC1	YES	YES
DEI reset	NOT TESTED	NOT TESTED

Hardware Overview

Below figures show the complete VPE Hardware.

VPE Driver Path As shown in below figures, the VPE 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



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 `VPS_M2M_INST_VPE1`,

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 fifth `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 have 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 BSP API Guide

System Init Phase

VPE M2M driver initialization happens as part of overall VPS 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.

VPS Init

```
Int32 Fvid2_init(Ptr args);
```

args - `NULL` currently not used.

```
Int32 Vps_init(const Vps_InitParams *initPrms);
```

initPrms - `Vps_InitParams *` VPS Initialization parameters.

```
    /* Init FVID2 */
    retVal = Fvid2_init(NULL);
    if (FVID2_SOK != retVal)
    {
        System_printf("FVID2 Init failed\n");
    }

    /* Init VPS */
    VpsInitParams_init(&vpsInitPrms);
    retVal = Vps_init(&vpsInitPrms);
    if (FVID2_SOK != retVal)
    {
        System_printf("VPS Init failed\n");
    }
```

Create Phase

In this phase user application opens or creates a driver instance. Each instance of the driver supports VPS_M2M_MAX_CH_PER_INST (defined in vps_m2m.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_DRV` to open the driver.

instanceId - `VPS_M2M_INST_VPE1` macro to open VPE1 memory driver.

createArgs - Pointer to `Vps_M2mCreateParams` structure containing valid create params. This parameter should not be NULL.

createStatusArgs - Pointer to `Vps_M2mCreateStatus` 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_M2mVpeChParams      chPrms;
    Vps_M2mCreateParams     createPrms;
    Vps_M2mCreateStatus     createStatus;

    /* Init create params */
    VpsM2mCreateParams_init(&createPrms);
    createPrms.numCh           = 1;
    createPrms.chInQueueLength = VPS_M2M_DEF_QUEUE_LEN_PER_CH;
    createPrms.isDeiFmdEnable  = FALSE;

    /* Init callback parameters */
    Fvid2CbParams_init(&cbPrms);
    cbPrms.cbFxn     = &App_m2mVpeAppCbFxn;
    cbPrms.errCbFxn  = &App_m2mVpeAppErrCbFxn;
    cbPrms.errList   = &errProcessList;
    cbPrms.appData   = appObj;

    /* Open the driver */
    fvidHandle = Fvid2_create(
```

```
                    FVID2_VPS_M2M_DRV,
                    VPS_M2M_INST_VPE1,
                    &createPrms,
                    &createStatus,
                    &cbPrms);
    if (NULL == fvidHandle)
    {
        System_printf("Create failed!!\n");
    }
```

FVID2 Control - Set VPE Parameters

`IOCTL_VPS_M2M_SET_VPE_PARAMS` IOCTL can be used to set the VPE hardware specific parameters. This IOCTL should be called after creating VPE M2M driver instance and before queueing or starting the M2M driver. Starting the M2M driver without calling this IOCTL will result in error. 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_M2M_SET_VPE_PARAMS` ioctl.

cmdArgs - Pointer to `Vps_M2mVpeParams` structure. This parameter should not be NULL.

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

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 VPE1 scalar, `scalarId` should be set to `VPS_M2M_VPE_SCALER_ID_SC0`.

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

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 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 BSP 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

VPE DEI/Scale

This example illustrates the VPE SC/DEI operation supportted by VPE memory to memory driver

- VPE Instance: VPS_M2M_INST_VPE1

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 output path.

- Please refer Common Steps for connecting CCS, running gel file etc.
- Load *bsp_examples_m2mVpeScale_m4vpss_release.xem4* executable file found at *$(rel_folder)\binary\bsp_examples_m2mVpeScale\bin\tda2sedx-virtio*

to IPU1 Core 0 M4 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);
```

- 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

UserGuideBspDisplayDriver

Display Drivers

Introduction

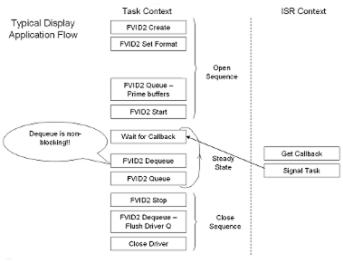
Display drivers takes the video buffer from the application and displays the video on the video encoder (VENC) at specified frame rate and resolution.

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.
- 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.
- 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 pipelines, Overlay Managers(LCD1,LCD2,LCD3,TV),display ports(DPI1,DPI2,DPI3) modules statically and dynamically (but not at run time, i.e. after display is started)
- All VENCs(LCD1,LCD2,LCD3) support upto 720p60, 1080p30, 1080i60 and 1080p60 mode
- Supports FVID2 interface

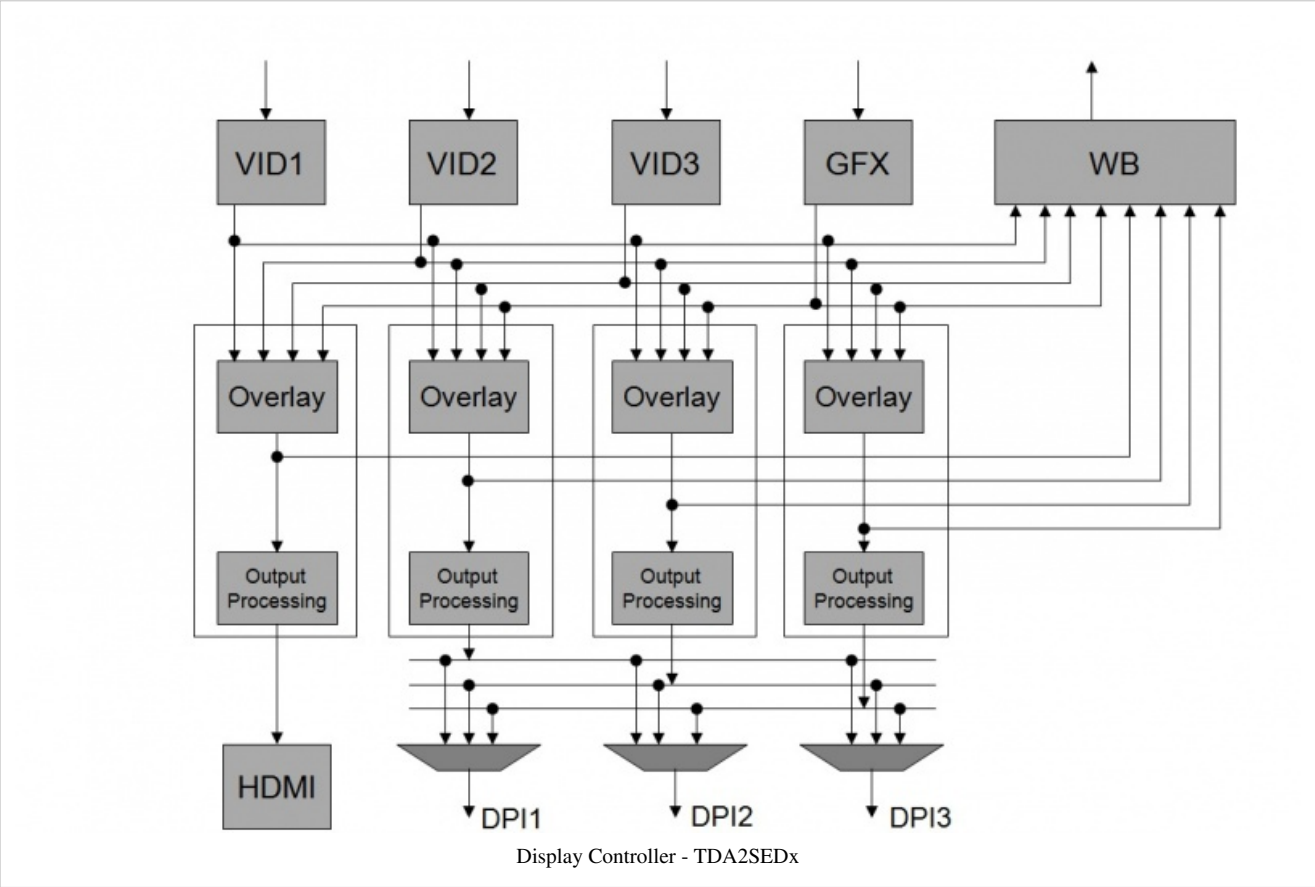
Features Not Supported

- Run time changing of Overlay manager parameters once it has started

Hardware Overview

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

Overview - TDA2SEDx



Display Controller - TDA2SEDx

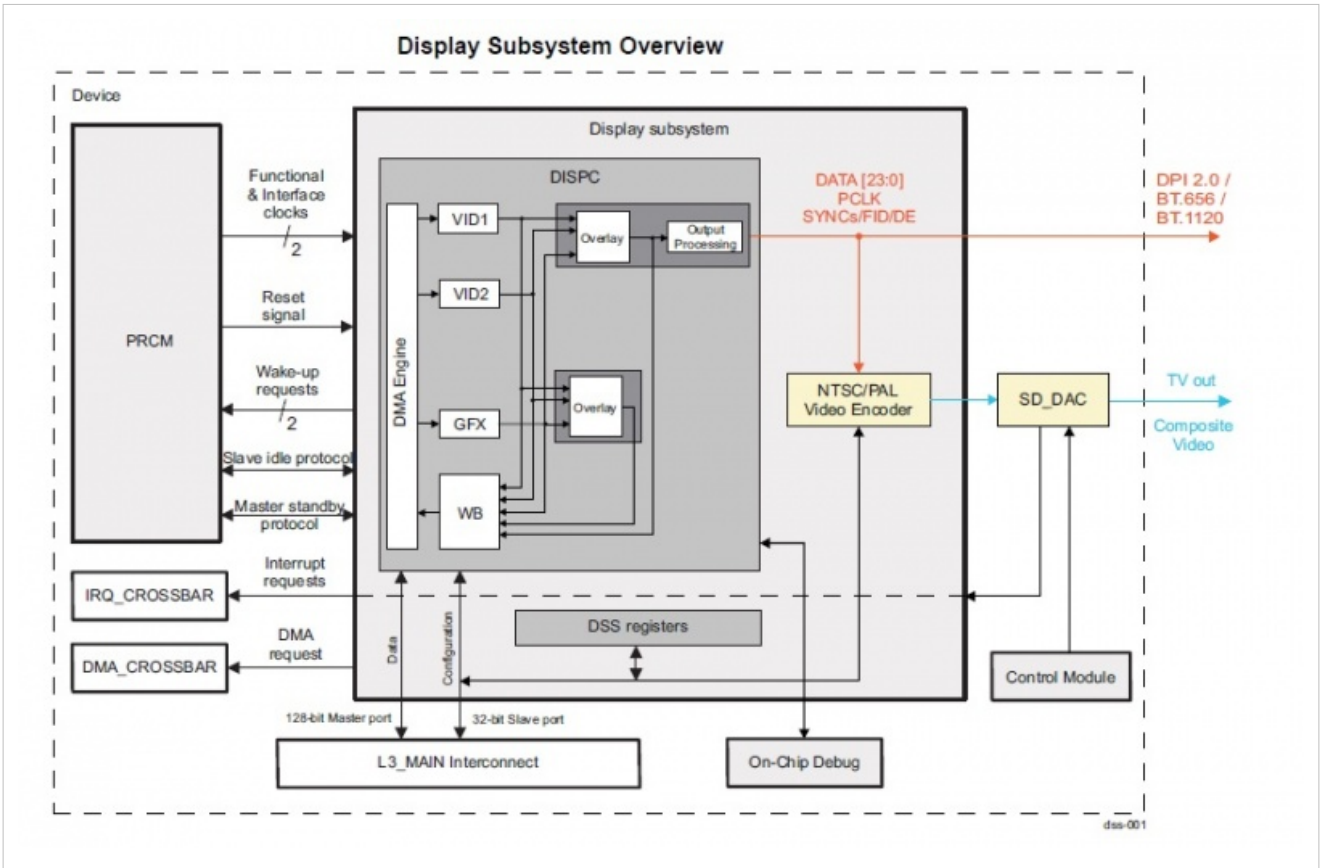
As shown in the figure, display controller driver controls configuration of paths i.e connnections of pipelines(VID1,VID2,VId3,GFX), Overlay managers(LCD1, LCD2,LCD3) and Display ports(DPI1,DPi2,DPI3). It also configures the timing of overlay managers.

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.

Overview - TDA3xx

Figure shows a block diagram with the DSS module within the device.



The supported display interfaces are:

- One parallel CMOS output, which can be used for MIPI® DPI 2.0, or BT-656 or BT-1120.
- One TV output, which is connected to the internal Video Encoder module (VENC). The VENC drives a single video digital-to-analog converter (SD_DAC) supporting composite video mode.

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_DctrlConfig`. This structure 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_DctrlConfig` 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_DctrlConfig`. This structure 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_DctrlConfig` 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 TDA2SEDx	Supported in TDA3xx
YUV422 interleaved format only for Video pipeline	YES	YES
YUV420 semi planar format only for video pipeline	Yes	YES
YUV422 semi planar format only for video pipeline	No	No
BGRX_4444,XBGR_4444,AGBR16_4444,RGBA16_4444,XGBR16_1555,AGBR16_1555,BGR16_565,XBGR24_8888,RGBX24_8888,ABGR32_8888,RGBA32_8888,BGR88,BGRA888,BGR565 data formats for both video and Graphics pipelines	YES	YES
Resolution upto 1080P@60FPS display	YES	YES
NTSC interlaced display on DPI Display Ports	YES	YES
Non-blocking queue/dequeue operation	YES	YES
Periodic callback feature	YES	YES
Field merged interlaced buffer mode	YES (tested on Zebu)	YES
Field separated interlaced buffer mode	YES (tested on Zebu)	YES
Inline scaling of input buffer	YES	YES

- DSS Bypass mode - DSS can be configured to operate in bypass mode where no processing is done on the input data, DPI output will be bit exact with the input data.For configuration check DSS Bypass Mode Configuration

Features Not Supported

- Hardware Mosaic
- Run time change of resolution and position

Dctrl Features Supported

Following are the features supported

- Full screen mode
- Blending of video and graphics layers
- Source and transparency color keying
- Global Alpha blending

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 pipeline(video/ Graphics) paths as listed in detail in BSP 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_DSS_VID1` macro to open Video 1 path display driver or pass `VPS_DISP_INST_DSS_VID2` macro to open Video 2 path display driver or `VPS_DISP_INST_DSS_VID3` macro to open Video 3 path display driver or `VPS_DISP_INST_DSS_GFX1` macro to open Graphics 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_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 - Set Dss params

`IOCTL_VPS_DISP_SET_DSS_PARAMS` ioctl is used to Set the all parameters related to DSS. This is a blocking call.

```
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_DISP_SET_DSS_PARAMS` ioctl.

cmdArgs - Pointer to `Vps_DispDssParams`.

cmdStatusArgs - This Parameter can be 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.

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

Dss Display

This example illustrates the display streaming feature of video and graphics Path displaying buffers on LCD and TV outputs.

- Load *bsp_examples_displayDss_m4vpss_release.xem4* executable file found at *$(rel_folder)\binary\bsp_examples_DisplayDss\bin\$platform\example-name-release.xem4* to DSS M4 debug session
- Run the application and it will halt for the user to provide the Option to run.
- Then it will halt for the user to load the input frames.
- Using loadRaw command in script console of CCS, load 4 frames of 800 x 480 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 Buffers one after the other on LCD/TV.
- Application will stop after displaying `TOTAL_LOOP_COUNT` frames
- 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 `BUFFER_WIDTH` and `BUFFER_HEIGHT` macro according to the input buffer dimension. The application will automatically change the window sizes according to the buffer size and pitch.
- Option 3 and 4 has features of blending and transparency color keying. In option 3 gfx layer is blended with video layer.In option 4 two video planes(side by side) are blended with graphics layer on top of it.
- Option 5 showcases use of all four pipelines(three video and one graphics), first video pipeline output is of full screen, the other two pipeline outputs are of small size window. These three video pipelines are blended with full size graphics layer.

DSS Bypass Mode

DSS (Display sub system) in Tda2xx/Tda3xx can be configured to operate in bypass/Transparent mode. It means that there will be no processing on the input data. It will be sent out on the DPI output as is.

This mode can be used to transfer data from DMA to other modules via DPI port.

The IOCTL call should be made with following parameters.

`IOCTL_VPS_DISP_SET_DSS_PARAMS`

- dispDssPrms.inFmt.width = Should be same as Overlay width.
- dispDssPrms.inFmt.height = Should be same as Overlay Height.
- dispDssPrms.inFmt.pitch[0] = (dispDssPrms.inFmt.width * 3)
- dispDssPrms.inFmt.dataFormat = FVID2_DF_BGR24_888
- dispDssPrms.inFmt.scanFormat = FVID2_SF_PROGRESSIVE
- dispDssPrms.tarWidth = dispDssPrms.inFmt.width;
- dispDssPrms.tarHeight = dispDssPrms.inFmt.height;
- dispDssPrms.posX = 0;
- dispDssPrms.posY = 0;
- dispDssPrms.memType = VPS_VPDMA_MT_NONTILEDMEM;
- dispDssPrms.vidCfg->pipeCfg.repliEnable = FALSE;
- dispDssPrms.vidCfg->pipeCfg.scEnable = FALSE;
- dispDssPrms.vidCfg->pipeCfg.cscFullRngEnable = VPS_DSS_DISPC_CSC_FULL;
- dispDssPrms.vidCfg->pipeCfg.chromaSampling = 0;
- dispDssPrms.vidCfg->pipeCfg.vc1Cfg->enable = FALSE;
- dispDssPrms.vidCfg->pipeCfg.vc1Cfg->rangeY = 0;
- dispDssPrms.vidCfg->pipeCfg.vc1Cfg->rangeUV = 0;
- dispDssPrms.vidCfg->pipeCfg.advDmaCfg = NULL;
- dispDssPrms.vidCfg->pipeCfg.gfxCfg = NULL;

`IOCTL_VPS_DCTRL_SET_CONFIG` in case of Display Controller driver

- mInfo = &dctrlCfg->vencInfo.modeInfo[0U].mInfo;
- mInfo->standard = FVID2_STD_CUSTOM;

- mInfo->width = overlay Width; /*Same width should be set for dispDssPrms.inFmt.width */
- mInfo->height = overlay height; /*Same height should be set for dispDssPrms.inFmt.height */
- mInfo->scanFormat = FVID2_SF_PROGRESSIVE;

The timing like hsynclength, vsync length and porches should be as per the width and height.

In `IOCTL_VPS_DCTRL_SET_OVLY_PARAMS IOCTL`.

Both alpha blending and color key should be disabled.

- panelCfg->alphaBlenderEnable = 0;
- panelCfg->colorKeyEnable = 0;

Article Sources and Contributors

BSP UserGuide *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=210747 *Contributors*: A0131716, A0132007, A0132235, A0132325, Csghone, Jags269, Mythripk, SivarajR, SujithShivalingappa, X0102720, X0135127, X0153534, X0190988

UserGuideBSPFolderOrg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=164230 *Contributors*: X0102720, X0135127

Vayu-Overview *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=211030 *Contributors*: X0102720, X0135127, X0190988

TDA3XX-Overview *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=211033 *Contributors*: X0135127, X0190988

BSP-Software Overview *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=145342 *Contributors*: X0102720

UserGuideFVID2 Vayu *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=170660 *Contributors*: A0131716

UserGuideBSPPlatformAPIs *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=183620 *Contributors*: SivarajR, X0102720

UserGuideBSPCaptureDriver *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=191202 *Contributors*: SivarajR, X0102720, X0135127

UserGuideBSPM2mVpeDriver *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=155336 *Contributors*: SivarajR

UserGuideBspDisplayDriver *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=212368 *Contributors*: A0131716, X0153534

Image Sources, Licenses and Contributors

Image:BSPInstall 1.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_1.PNG *License*: unknown *Contributors*: X0102720

Image:BSPInstall 2.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_2.PNG *License*: unknown *Contributors*: X0102720

Image:BSPInstall 3.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_3.PNG *License*: unknown *Contributors*: X0102720

Image:BSPInstall 4.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_4.PNG *License*: unknown *Contributors*: X0102720

Image:BSPInstall 5.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_5.PNG *License*: unknown *Contributors*: X0102720

Image:BSPInstall 6.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_6.PNG *License*: unknown *Contributors*: X0102720

Image:BSPInstall 7.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_7.PNG *License*: unknown *Contributors*: X0102720

Image:BSPInstall 8.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_8.PNG *License*: unknown *Contributors*: X0102720

Image:BSPInstall 9.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPInstall_9.PNG *License*: unknown *Contributors*: X0102720

File:TDA3xx_Multi_Deserializer_Jumper_Setup.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TDA3xx_Multi_Deserializer_Jumper_Setup.jpg *License*: unknown *Contributors*: SivarajR

Image:mcspitda3xxJ28.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Mcspitda3xxJ28.jpg *License*: unknown *Contributors*: X0190988

Image:mcspitda3xxConnector.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Mcspitda3xxConnector.jpg *License*: unknown *Contributors*: X0190988

Image:mcspiloopbackConnection.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:McspiloopbackConnection.jpg *License*: unknown *Contributors*: X0190988

File:TDA2xx_Multi_Deserializer_Jumper_Setup.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TDA2xx_Multi_Deserializer_Jumper_Setup.jpg *License*: unknown *Contributors*: SivarajR

Image:mcspitda2xxconnect.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Mcspitda2xxconnect.jpg *License*: unknown *Contributors*: X0190988

Image:mcspitda2xxloopbackConnection.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Mcspitda2xxloopbackConnection.jpg *License*: unknown *Contributors*: X0190988

Image:CPLD Programming Step 5.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:CPLD_Programming_Step_5.png *License*: unknown *Contributors*: A0132235

Image:CPLD Programming Step 6.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:CPLD_Programming_Step_6.png *License*: unknown *Contributors*: A0132235

Image:CPLD Programming Step 8.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:CPLD_Programming_Step_8.png *License*: unknown *Contributors*: A0132235

File:TDA3xx_EVM_TOP_BSP.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TDA3xx_EVM_TOP_BSP.jpg *License*: unknown *Contributors*: SujithShivalingappa

File:TDA3xx_EVM_CSI2.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TDA3xx_EVM_CSI2.jpg *License*: unknown *Contributors*: SujithShivalingappa

File:TDA3xx_EVM_TOP_updated.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:TDA3xx_EVM_TOP_updated.jpg *License*: unknown *Contributors*: SujithShivalingappa

File:BSPTopLevel.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPTopLevel.PNG *License*: unknown *Contributors*: X0102720

File:BSPBuildFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPBuildFolder.PNG *License*: unknown *Contributors*: X0102720

File:BSPDocsFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPDocsFolder.PNG *License*: unknown *Contributors*: X0102720

File:BSPMakeFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPMakeFolder.PNG *License*: unknown *Contributors*: X0102720

File:BSPPackagesFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPPackagesFolder.PNG *License*: unknown *Contributors*: X0102720

File:BSPCommonFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPCommonFolder.PNG *License*: unknown *Contributors*: X0102720

File:BSPExamplesFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPExamplesFolder.PNG *License*: unknown *Contributors*: X0102720

File:BSPVpsFolder.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPVpsFolder.PNG *License*: unknown *Contributors*: X0102720

Image:Vip_overview.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Vip_overview.jpg *License*: unknown *Contributors*: X0190988

Image:VPE Blockdiagram.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:VPE_Blockdiagram.jpg *License*: unknown *Contributors*: X0190988

Image:DSS_overview.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DSS_overview.jpg *License*: unknown *Contributors*: X0190988

Image:VIP_overview_Tda3xx.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:VIP_overview_Tda3xx.jpg *License*: unknown *Contributors*: X0190988

Image:DSS_overview_Tda3xx.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DSS_overview_Tda3xx.jpg *License*: unknown *Contributors*: X0190988

Image:ISS_overview.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:ISS_overview.jpg *License*: unknown *Contributors*: X0190988

Image:BSPBlockDiagram.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPBlockDiagram.PNG *License*: unknown *Contributors*: X0102720

Image:BSPDriverStack.PNG *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:BSPDriverStack.PNG *License*: unknown *Contributors*: X0102720

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_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: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:Frame Drop Mode Buffer Flow (2).png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Frame_Drop_Mode_Buffer_Flow_(2).png *License*: unknown *Contributors*: X0135127

Image:Frame Drop Mode ISR Flow Diagram.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Frame_Drop_Mode_ISR_Flow_Diagram.png *License*: unknown *Contributors*: X0135127

Image:Last Frame Repeat Mode Buffer Flow.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Last_Frame_Repeat_Mode_Buffer_Flow.png *License*: unknown *Contributors*: X0135127

Image:Last Frame Repeat Mode ISR Flow Diagram.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Last_Frame_Repeat_Mode_ISR_Flow_Diagram.png *License*: unknown *Contributors*: X0135127

Image:Circular Frame Repeat Mode Buffer Flow.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Circular_Frame_Repeat_Mode_Buffer_Flow.png *License*: unknown *Contributors*: X0135127

Image:Circular Frame Repeat Mode ISR Flow Diagram.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Circular_Frame_Repeat_Mode_ISR_Flow_Diagram.png *License*: unknown *Contributors*: X0135127

Image: VPE1_HW_BlockDiagram.jpeg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:VPE1_HW_BlockDiagram.jpeg *License*: unknown *Contributors*: SivarajR

Image:DisplayFlowChartVayu.png *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:DisplayFlowChartVayu.png *License*: unknown *Contributors*: A0131716

Image:Dctrl_Dss_Vayu.jpg *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Dctrl_Dss_Vayu.jpg *License*: unknown *Contributors*: A0131716