# DM648 VPORT Driver

# USER'S GUIDE

**Document Revision History**

| Rev No | Author(s) | Revision History | Date | Approval(s) |
|---|---|---|---|---|
| 0.1 | Sivaraj R<br>Grishma Parikh | Corrected some API descriptions | May 7, 2007 | Initial Draft |
| 0.2 | Vichu | Formatted to new template and corrected FVID API section | May 9, 2007 | Draft |
| 0.3 | Sivaraj R | Added HD configuration and new sample applications.<br>Updated for patch 1.10.00.00 | June 15, 2007 | Draft |
| 0.4 | Sivaraj R | Added HD loopback and VESA application sample application description. | July 10, 2007 | Draft |
| 0.5 | Grishma Parikh<br>Sivaraj R | Added description for slice mode capture IOCTLs and RAW capture support. Separate section is added to have detailed description for all IOCTLs | August 30, 2007 | Draft |
| 0.6 | Grishma Parikh | Driver Porting section added | September 13, 2007 | Draft |
| 0.7 | Sivaraj R | Changed XDC and BIOS versions in system requirement | October 22, 2007 | Draft |
| 0.8 | Sivaraj R | Changed directory structure according to new package | November 27, 2007 | Draft |
| 0.9 | Sivaraj R | Added description about THS7353 enable/disable macro in EDC driver | January 14, 2008 | Draft |
| 0.10 | Sivaraj R | Added TCI file driver initialization illustration and added dependent libraries for building video application | January 24, 2008 | Draft |
| 0.11 | Sivaraj R | Added description about HD capture workaround in production EVM | February 20, 2008 | Draft |
| 0.12 | Chandan Nath | Updated for adding compiler switches in build options | May 21, 2008 | Draft |

# TABLE OF CONTENTS

# TABLE OF FIGURES

# 1 Introduction

This document is the reference guide for the video port driver and it explains how to configure and use the driver.

DSP/BIOS applications use the driver typically through FVID APIs to perform frame video capture and display. FVID was implemented as a simple wrapper on top of the GIO class driver and provides an application-specific interface that has been customized for frame video. For more information on the DSP/BIOS device driver model and the GIO class driver, refer to the References section of this document.

**A NOTE ON COMPATIBLITY:** DM642 video drivers featured FVID APIs. For the compatibility with other video drivers (DM64LC) and for future use, some APIs have been added to the existing (DM642) FVID APIs set, without affecting the backward compatibility (This needs recompilation of the application with new APIs header and the new drivers) of DM642 applications. This document also describes the DM648s FVID API descriptions.

## 1.1 Terms & Abbreviations

| Term | Description |
|---|---|
| ☞ | This bullet indicates important information.<br>Please read such text carefully. |
| ❑ | This bullet indicates additional information. |
| Legacy Mode | DM642 API Compatibility mode |
| Normal Mode | DM64LC API Compatibility mode |

## 1.2 References

| | | |
|---|---|---|
| 1. | SPRA918A – August 2003 – Application Report | The TMS320DM642 Video Port Mini-Driver |
| 2. | SPRUEM1_Video_Port.pdf | DM648 Video Port Peripheral Reference Guide |

## 1.3 S/W Support

This VPORT device driver has been developed for the DSP/BIOS operating system using the TI supplied Chip Support Library. For more details on the version numbers refer to the release notes in the root of the installation.

## 1.4 Driver Components

The Video driver is constituted of following sub components:

**VPORT Driver –** application interface, VPORT and EDMA handling

**EDC (External Device Control) Driver –** Configures external Video Decoder and Encoder. VPORT driver library calls EDC Driver APIs for external Decoder and Encoder configurations

**System components:**

**PALOS –** DSP/BIOS Abstraction
Below Figure shows DM648 Video driver architecture.

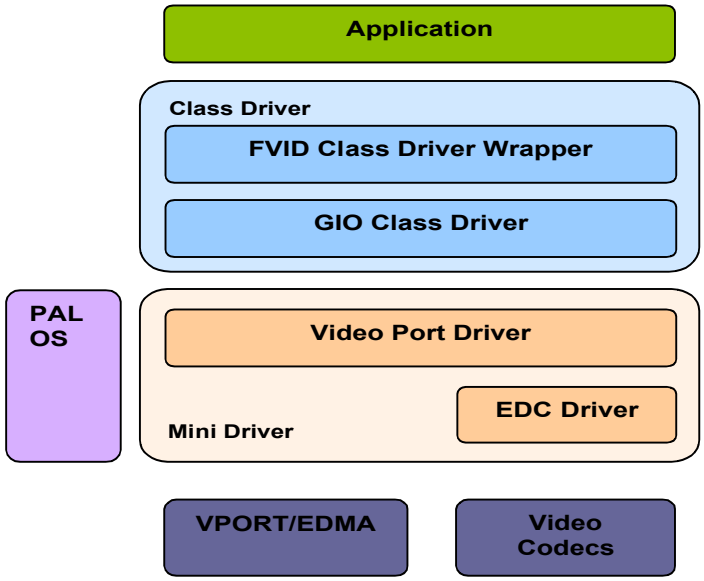


**Figure 1.** DM648 Video driver architecture

## 1.5 Default Driver Configuration

VPORT driver does not have any default configuration support. Before using the driver, application should configure the driver with valid configurations. In case the driver recognizes invalid configuration parameter it will return the corresponding error code.

All EDC drivers are having default configuration.

- TVP5154 decoders are configured for NTSC capture (with embedded sync – BT.656 mode).
- SAA7105 encoder is configured for NTSC display (with embedded sync – BT.656 mode) for SD display. And is put to power down state for HD display.
- If SAA7105 encoder is selected at the time of channel creation then THS8200 encoder is put in BT.656 bypass mode otherwise THS8200 encoder is by default configured for 1080i 60Hz display.
- TVP7000 decoder is by default configured for 1080i 60Hz capture.

## 1.6 Driver Capabilities

The significant driver features are:

- Supports Multiple Video Port Instances (4 captures and 1 display video port instances are supported on DM648 EVM)
- Supports dual channel 8-bit BT.656, single channel 16-bit Y/C, dual channel 8-bit RAW or single channel 16-bit RAW capture per Video Port
- Supports single channel 8-bit BT.656, single channel 16-bit Y/C, single channel 8-bit RAW or single channel 16-bit RAW display per Video Port
- Supports enable/disable of video port global interrupt on all defined video port events
- External Device Control Interface using EDC driver for seamless integration with different video encoder or decoder devices
- Supports flipping of multiple frame buffers for seamless capture and display operation
- Easy to maintain & re-target to new platforms

## 1.7 System Requirements

Details about the tools and the BIOS version that the driver is compatible with can be found in the system Release Notes.

# 2 Installation Guide

## 2.1 Component Folder

Upon installing the VPORT driver the following directory structure is found in the driver's directory.

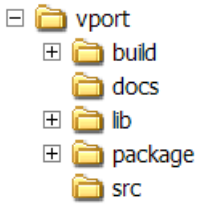


**Figure 2.** VPORT Driver Directory Structure

- **vport:** This top level vport folder contains vport driver psp header files and XDC package files (package.bld, package.xdc and package.xs).
- **build:** This folder contains VPORT driver library project file. The generated driver library shall be included in the application where VPORT driver have to be used.
- **docs:** This folder contains architecture document, datasheet, release notes and user guide.

  Architecture document contains the driver details which can be helpful for the developers as well as consumers to understand the driver design.

  Datasheet gives the idea about the memory consumption by the driver and description of the top level APIs.

  Release Note gives the details about system requirements, steps to Install/Uninstall the package. This document lists the known issues of the driver if any.

  User Guide provides information about how to use the driver. It contains description of sample applications which guide the end user to make their applications using this driver.
- **lib:** This folder contains libraries generated in all the configuration modes (debug, idebug, irelease and release)
- **package:** This folder contains files generated by XDC tool.
- **src:** This folder contains VPORT driver source files. It also contains header files that are used by the driver.

## 2.2 Build

This section describes for each supported target environment, the applicable build options, supported configurations and how to select the featured capabilities and how to enable the allowed user customizations for the software to be installed and how the same can be realized.

The component might be delivered to user in different formats:

- Source-less i.e. binary executables and object libraries only.
- Source-inclusive i.e.The entire source code used to implement the driver is included in the delivered product.
- Source-selective ie. Only a part of the overall source is included. This delivery mechanism might be required either because certain parts of the driver require soruce level extensions and/or customization at the user's end or because specific parts of the driver is exposed to user at the source level to insure user's software development.

When source is included as part of the product delivery, the CCS project file is provided as part of the package. When object format is distributed, the driver header files are part of the "*vport*\" folder and the driver library is provided in "*vport**lib"* folder.

## 2.3 Build Options

This driver does not have any specific build option.

The build folder contains a CCS project file that builds the driver into a library for debug and release mode.

Build options – *"iDebug"* and *"iRelease"* are provide to enable VPORT Instrumentation in the driver.

Following compiler switches are used to compile for different options.

- **_DEBUG**
  This is used as a flag to compiler whether to include the debug statements inserted in the code into the final image. This flag helps to build DEBUG image of the program. For RELEASE images this is not passed to the compiler.
  Reference: Project file for vport.vpfe etc
- **CHIP_XXXX**
  The CSL layer is written in a common file for all the variants of a SOC. This flag differentiates the variant we are compiling for, for e.g. - CHIP_DM648, and the CSL definitions for that variant appropriately gets defined for register base addresses, num of ports of a peripheral etc.
  Reference : soc\DM648\dsp\inc\csl_vport_vphal.h
- **VPORT_INSTRUMENTATION_ENABLED**
  This flag is passed to the compiler to include the instrumentation code parts into the final image/lib of the program. This helps build the iRelease/iDebug versions of the image/lib with a common code base

- **VPORT_DEBUG**
  (un)definition of this macro in _vport.h disables/enables selectively the debug statements for vport to be included into the final image/lib

# 3 DSP/BIOS VPORT DRIVER Structures

This section discusses about the initialization details and initialization structures used in the VPORT driver. Please note that for some structure member information/details, the DM648 video port peripheral reference guide might need to be referred.

Most members of these structures directly reflect the VPORT register settings. The driver **does not** check the validity of these parameters. It is the application's responsibility to pass proper value according to the VPORT register description. Kindly refer VPORT Peripheral Reference Guide for more details.

### 3.1.1 Initialization details

To use the capture or display VPORT device driver, a device entry must be added and configured in the DSP/BIOS configuration tool.

To have VPORT device driver included in the application, corresponding TCI file have to be included in BIOS TCF i.e. *"dm648_vport0.tci"*, *"dm648_vport1.tci", "dm648_vport2.tci", "dm648_vport3.tci" or "dm648_vport4.tci"* must be included in BIOS TCF file of the application for using VPORT 0, 1, 2, 3 or 4 instances of the driver respectively. These files can be found in video sample directory.

The following are the device configuration settings required to use the capture driver. Note: This has to be done for all of the required driver instances.

| TCI Configuration Parameters | Description |
|---|---|
| *initFxn* - Init Function | Pointer to application function to initialize DM648 video ports like module clock enabling and enabling pin muxing. |
| *fxnTable* - Function Table Pointer | *VPORTCAP_Fxns*. This is a global variable which points to the VPORT driver APIs. |
| *fxnTableType* - Function Table Type | *IOM_Fxns* |
| *deviceId* - Device Id | Specify which video port to use. For example to use VPORT 0 this should be given as 0. |
| *params* – Pointer to Port parameter | An optional pointer to an object of type *VPORT_PortParams* as defined in the header file *vport.h*. This pointer will point to a device parameter structure. In BIOS TCI files, this structure object is passed as an argument. Application should declare and initialize the structure object properly. |
| Device Global Data Pointer | N/A, not used by this driver |

The capture driver initialization in BIOS TCF looks like this (assumed VPORT 0 as capture port):

```
bios.VP0CAPTURE                 = bios.UDEV.create("VP0CAPTURE");
bios.VP0CAPTURE.initFxn         = prog.extern("VPORT0_init");
bios.VP0CAPTURE.fxnTable        = prog.extern("VPORTCAP_Fxns");
bios.VP0CAPTURE.fxnTableType    = "IOM_Fxns";
bios.VP0CAPTURE.params          = prog.extern("vCapParamsPort");
bios.VP0CAPTURE.deviceId        = 0x00;
```

The following are the device configuration settings required to use the display driver:
Note: This has to be done for all of the required driver instances.

| TCI Configuration Parameters | Description |
|---|---|
| *initFxn* - Init Function | Pointer to application function to initialize DM648 video ports like module clock enabling and enabling pin muxing. |
| *fxnTable* - Function Table Pointer | *VPORTDIS_Fxns*. This is a global variable which points to the VPORT driver APIs. |
| *fxnTableType* - Function Table Type | *IOM_Fxns* |
| *deviceId* - Device Id | Specify which video port to use. For example to use VPORT 1 this should be given as 1. |
| *params* – Pointer to Port parameter | Same as for the capture driver |
| Device Global Data Pointer | N/A, not used by this driver |

The display driver initialization in BIOS TCF looks like this (assumed VPORT 1 as display port):

```
bios.VP1DISPLAY                = bios.UDEV.create("VP1DISPLAY");
bios.VP1DISPLAY.initFxn        = prog.extern("VPORT1_init");
bios.VP1DISPLAY.fxnTable       = prog.extern("VPORTDIS_Fxns");
bios.VP1DISPLAY.fxnTableType = "IOM_Fxns";
bios.VP1DISPLAY.params         = prog.extern("vDisParamsPort");
bios.VP1DISPLAY.deviceId       = 0x01;
```

Apart from the VPORT driver initialization, I2C driver should also be initialized in the BIOS TCF file. For details on how to initialize I2C driver, refer I2C driver user guide – *BIOS_I2C_Driver_UserGuide.pdf*.

### 3.1.2 VPORT_PortParams

*"vport.h"* file contains *VPORT_PortParams* data structure that is passed while mdBindDev call which is defined with UDEV VPORT parameters in TCF file of application. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *dualChanEnable* | Boolean:<br>*TRUE* = Dual channels enabled for capture<br>*FALSE* = Single channel enabled for capture or display. For display driver this parameter should be always *FALSE*. |
| *vc1Polarity* | VPORT control pin 1 polarity. This value should be *VPORT_POLARITY_ACTIVE_HIGH* or *VPORT_POLARITY_ACTIVE_LOW* depending on required operation. |
| *vc2Polarity* | VPORT control pin 2 polarity. Same as *vc1Polarity*. |
| *vc3Polarity* | VPORT control pin 3 polarity. Same as *vc1Polarity*. |
| *edcTbl* | Array of function tables of EDC module (refer to section External Device Control section for details) |

### 3.1.3 VPORT_VIntCbParams

*"vport.h"* file contains *VPORT_VIntCbParams* data structure that is passed in *VPORT_CMD_SET_VINTCB* IOCTL to configure VPORT interrupts. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *cbArg* | Call-back argument is for the application to identify which channel or device causes the interrupt |
| *vIntCbFxn* | Pointer to the video port interrupt call-back function of type *VPORT_IntCallBack*. |
| *vIntMask* | Interrupt mask. This determines the interrupts for which the application will be notified using the registered call back function. Refer "*vport.h*". All masks should be bitwise OR together to indicate errors that need to be handled by the call-back. By default the driver handles display under run and capture over run errors. Application doesn't have to do any recovery if these errors occur. But this will be indicated to the application if call backs are registered. |
| *vIntLine* | Line number where vertical interrupt should occur when *VPORT_INT_VINT1* and *VPORT_INT_VINT2* are enabled. Don't care otherwise |
| *irqId* | Hardware interrupt id at which Video Port Interrupt occurs. This value is don't care in the current driver as the driver assumes the IRQ number for the Video Port – **"8"**. If the application changes the ECM mapping then the macro defined in *"vport.h"* should be changed accordingly and driver should be recompiled. This is given to support backward compatibility. |

### 3.1.4 VPORT_CurrParams

*"vport.h"* file contains *VPORT_CurrParams* data structure that is passed while *VPORT_CMD_GET_PARAMS* IOCTL call. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *mode* | Current video mode of driver – 8 bit BT.656, 8 bit YCbCr, 8/16 bit Raw modes. |
| *fldOp* | Field or frame operation. Possible values are flied 1 only, field 2 only, frame (field 1 + field 2) and progressive |
| *scale* | Boolean.<br>*TRUE* = Horizontal scaling enabled.<br>*FALSE* = Horizontal scaling disabled. |
| *resmpl* | Boolean.<br>*TRUE* = Chroma re-sampling enabled.<br>*FALSE* = Chroma re-sampling disabled. |
| *yPitch* | Luminance line pitch in bytes. This need not always be equal to the display or capture line size. Due to hardware alignment requirements, this will be a multiple of 8 bytes. |
| *cPitch* | Chrominance line pitch in bytes. This need not always be |

| | |
|---|---|
| | equal to the display or capture chrominance line size. Due to hardware alignment requirements, this will be a multiple of 8 bytes. |
| *numLines* | Total number of lines per frame |
| *numPixels* | Number of pixel in one line |
| *bufSize* | Size of one FVID frame buffer |
| *mergeFlds* | Field operation. Boolean:<br>*TRUE* = Fields are merged in interlaced operation<br>*FALSE* = Fields are separated in interlaced operation.<br>For progressive modes, this is always *FALSE.* |

### 3.1.5 VPORTCAP_Params

*"vportcap.h"* file contains *VPORTCAP_Params* data structure that is passed while *FVID_create* call. Most members of this structure directly reflect the VPORT register settings. The driver **does not** check the validity of these parameters (Example *cmode*, *fldOp* etc). Kindly refer VPORT Peripheral Reference Guide for more details. The values to be used for most of the members are given in *"vport.h"* and *"vportcap.h"* files. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *cmode* | Capture mode settings – 8 bit BT.656, 8 bit YCbCr, 8/16 bit Raw modes. 10 and 20 bit modes are not supported by the VPORT and driver |
| *fldOp* | Field & frame operation. Possible values are flied 1 only, field 2 only, frame (field 1 + field 2) and progressive |
| *scale* | Boolean.<br>*TRUE* = Horizontal scaling enabled.<br>*FALSE* = Horizontal scaling disabled.<br>Indicates whether to enable horizontal ½ scaling. The ½-scaling mode is used to reduce the horizontal resolution of captured luminance and chrominance data by a factor of two. Note: Vertical ½ scaling should be done by application if required. When scaling is enabled make sure the Luminance and C line sizes are multiple of 8 for proper driver operation |
| *resmpl* | Boolean.<br>*TRUE* = Chroma re-sampling enabled<br>*FALSE* = Chroma re-sampling disabled |
| *bpk10Bit* | Bit pack mode. Don't care as 10 bit modes are not supported |
| *hCtRst* | Horizontal counter reset mode |
| *vCtRst* | Vertical counter reset mode |
| *fldDect* | Enable whether to use FID input or field detection logic based on the timing relation of hsync and vsync |
| *extCtl* | Enable external timing control |
| *fldInv* | Enable inversion of the detected field |
| *fldXStrt1* | Field 1 X start |

| | |
|---|---|
| *fldYStrt1* | Field 1 Y start |
| *fldXStrt2* | Field 2 X start |
| *fldYStrt2* | Field 2 Y start |
| *fldXStop1* | Field 1 X stop |
| *fldYStop1* | Field 1 Y stop |
| *fldXStop2* | Field 2 X stop |
| *fldYStop2* | Field 2 Y stop |
| *thrld* | Video FIFO threshold in double words. This value is normally equivalent to line size divided by 8. Represents number of double words of Luminance data that are transferred per EDMA event. C threshold is half the Y threshold if Y threshold is even else will round to the nearest integer. When half line size or quarter line size threshold are provided (as required for HDTV and VESA modes), care should be taken to make sure that the Y and C line size are multiple of threshold. If not the driver will assume the threshold to be one line size. |
| *numFrmBufs* | Number of frame buffers that the driver allocates. If this is 0, then the driver operates in Normal mode. When this value is other than 0, then driver operates in Legacy mode. In both cases, minimum of 3 buffers are recommended for proper driver operation |
| *alignment* | Frame buffer alignment. Used at the time of allocating buffer |
| *mergeFlds* | Field operation. Boolean:<br>*TRUE* = Fields are merged in interlaced operation<br>*FALSE* = Fields are separated in interlaced operation.<br>For progressive modes, this should be always *FALSE* |
| *segId* | Memory segment ID, used at the time of buffer allocation |
| *autoSyncEnable* | Boolean to enable auto sync operation for field operation. When this is enabled, the driver will synchronize to field 1 for every frame. When the field order changes (this may occur when the input cable is removed and connected again), the driver reconfigures the EDMA parameters to properly capture field 1 first and then field 2. If this is not done, the captured frame may contain field 2 of previous frame and field 1 of next frame. |
| *hEdma* | EDMA3 driver handle |

### 3.1.6 VPORTCAP_ParamsRaw

*"vportcap.h"* file contains *VPORTCAP_ParamsRaw* data structure that is passed while *FVID_create* call if raw mode capture is required. The driver **does not** check the validity for the parameters related to VPORT registers. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *cmode* | Capture mode settings – This should be VPORT_MODE_RAW_8BIT for 8-bit RAW capture or VPORT_MODE_RAW_16BIT for 16-bit RAW capture |
| *bpk10Bit* | 10-bit bit-pack mode as defined by video port. Don't care as 10 bit modes are not supported |
| *startupSyncEnable* | Boolean to enable start up sync or not |
| *blankperiod* | Minimum time for CAPEN signal to stay inactive before interpret it as vertical blanking only used when SSE is set |
| *lineSz* | Number of pixels per line |
| *numLines* | Number of lines per frame |
| *thrld* | FIFO threshold value indicates number of double words to generate DMA events. This value is normally equivalent to line size divided by 8. Represents number of double words of data that are transferred per EDMA event. When half line size or quarter line size threshold are provided, care should be taken to make sure that the line size is multiple of threshold. If not the driver will assume the threshold to be one line size. |
| *numFrmBufs* | Number of frame buffers to be used by driver. Minimum of 3 buffers are recommended for proper operation. |
| *alignment* | Frame buffer alignment |
| *segId* | Memory segment ID, used by driver to allocate video frame buffer |
| *hEdma* | EDMA3 driver handle |

### 3.1.7 VPORTCAP_LinesCapturedInfo

*"vportcap.h"* file contains *VPORTCAP_LinesCapturedInfo* data structure that is passed with line mode interrupt callback function. This structure is also used as an argument in *VPORTCAP_CMD_GET_NUMLINES_CAPTURED* IOCTL. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *frmPtr* | *FVID_Frame* pointer which is currently captured |
| *numLines* | Number of lines captured in current frame |

### 3.1.8 VPORTCAP_LineIntParam

*"vportcap.h"* file contains *VPORTCAP_LineIntParam* data structure that is passed with *VPORTCAP_CMD_SET_LINE_INT* IOCTL call to set line mode interrupt. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *callBackFxn* | Callback function pointer to be called for line mode interrupt. This pointer is of type *VPORTCAP_LinesCapturedCB*. |
| *vIntFld1* | Line number at which interrupt occurs in field1 |
| *vIntFld2* | Line number at which interrupt occurs in field2 |
| *cbArg* | Call-back argument is for the application to identify which channel or device causes the interrupt |
| *errCheckEnable* | Enable short field detect error checking in EDMA ISR. This should be set to TRUE to handle short field detect error. Generally SFD error occurs when cable is plugged out/in or when driver is configured for PAL and a NTSC input is connected.<br>In normal scenario, this error handling is taken care in *FVID_dequeue(FVID_alloc)/FVID_queue(FVID_free)* call to driver. But when application is using slice mode capture, then there is no call to the driver. Hence this flag should be set so that the error is handled in EDMA ISR context. |

### 3.1.9 VPORTDIS_Params

*"vportdis.h"* file contains *VPORTDIS_Params* data structure that is passed while FVID_create call. Most of the members of this structure directly reflect the VPORT register settings. The driver **does not** check the validity of these parameters (Example *cmode*, *fldOp* etc). Kindly refer VPORT Peripheral Reference Guide for more details. The values to be used for most of the members are given in *"vport.h"* and *"vportdis.h"* files. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *dmode* | Display mode settings – 8 bit BT.656, 8 bit YCbCr, 8/16 bit Raw modes. 10 and 20 bit modes are not supported by the VPORT and driver |
| *fldOp* | Field & frame operation. Possible values are flied 1 only, field 2 only, frame (field 1 + field 2) and progressive |
| *scale* | Boolean.<br>*TRUE* = Horizontal scaling enabled.<br>*FALSE* = Horizontal scaling disabled.<br>Indicates whether to enable horizontal 2x scaling. The 2x scaling mode is used to increase the horizontal resolution of displayed luminance and chrominance data by a factor of two. Note: Vertical 2x scaling should be done by application if required. When scaling is enabled make sure the Luminance and Chrominance line sizes are multiple of 8 for proper driver operation |
| *resmpl* | Boolean.<br>*TRUE* = Chroma re-sampling enabled |

| | |
|---|---|
| | *FALSE* = Chroma re-sampling disabled |
| *defValEn* | Boolean.<br>*TRUE* = Default value enabled. *yDefVal, cbDefVal, crDefVal* values will be displayed during non-active frame area<br>*FALSE* = Default value disabled. Blanking values will be used during no-active frame area |
| *bpk10Bit* | 10-bit bit-pack mode. This is a don't care as 10 bit modes are not supported |
| *vctl1Config* | VCTL1 pin selection (Please refer to vportdis.h for available values) |
| *vctl2Config* | VCTL2 pin selection (Please refer to vportdis.h for available values) |
| *vctl3Config* | VCTL3 pin selection (Please refer to vportdis.h for available values) |
| *extCtl* | Enable external timing control. The current driver doesn't support external timing control of operation and hence this should be always disabled. |
| *frmHSize* | Frame horizontal size |
| *frmVSize* | Frame vertical size |
| *imgHOffsetFld1* | Image horizontal offset field 1 |
| *imgVOffsetFld1* | Image vertical offset field 1 |
| *imgHSizeFld1* | Image line size field 1 |
| *imgVSizeFld1* | Image total lines field 1 |
| *imgHOffsetFld2* | Image horizontal offset field 2 |
| *imgVOffsetFld2* | Image vertical offset field 2 |
| *imgHSizeFld2* | Image line size field 2 |
| *imgVSizeFld2* | Image total lines field 2 |
| *hBlnkStart* | Horizontal blanking start value |
| *hBlnkStop* | Horizontal blanking stop value |
| *vBlnkXStartFld1* | Vertical blanking pixel start value field 1 |
| *vBlnkYStartFld1* | Vertical blanking line start value field 1 |
| *vBlnkXStopFld1* | Vertical blanking pixel stop value field 1 |
| *vBlnkYStopFld1* | Vertical blanking line stop value field 1 |
| *vBlnkXStartFld2* | Vertical blanking pixel start value field 2 |
| *vBlnkYStartFld2* | Vertical blanking line start value field 2 |
| *vBlnkXStopFld2* | Vertical blanking pixel stop value field 2 |
| *vBlnkYStopFld2* | Vertical blanking line stop value field 2 |
| *xStartFld1* | Pixel start field 1 |
| *yStartFld1* | Line start field 1 |

| | |
|---|---|
| *xStartFld2* | Pixel start field 2 |
| *yStartFld2* | Line start field 2 |
| *hSyncStart* | Horizontal sync start |
| *hSyncStop* | Horizontal sync stop |
| *vSyncXStartFld1* | Vertical sync pixel start field 1 |
| *vSyncYStartFld1* | Vertical sync line start field 1 |
| *vSyncXStopFld1* | Vertical sync pixel stop field 1 |
| *vSyncYStopFld1* | Vertical sync line stop field 1 |
| *vSyncXStartFld2* | Vertical sync pixel start field 2 |
| *vSyncYStartFld2* | Vertical sync line start field 2 |
| *vSyncXStopFld2* | Vertical sync pixel stop field 2 |
| *vSyncYStopFld2* | Vertical sync line stop field 2 |
| *yClipLow* | Luminance (Y) low clipping value |
| *yClipHigh* | Luminance (Y) high clipping value |
| *cClipLow* | Chrominance (CB/CR) low clipping value |
| *cClipHigh* | Chrominance (CB/CR) high clipping value |
| *yDefVal* | Luminance (Y) default value for non-RAW mode operation. For RAW mode display, this represents bits 7-0 of the default value. |
| *cbDefVal* | Chrominance (CB) default value for non-RAW mode operation. For RAW mode display, this represents bits 15-8 of the default value. |
| *crDefVal* | Chrominance (CR) default value for non-RAW mode operation. For RAW mode display, this represents bits 19-16 of the default value. |
| *rgbX* | RGB extract enable/disable select. Used in RAW mode display operation (1 enable, 0 disable) |
| *incPix* | Pixel increment for RAW mode only (0 otherwise) |
| *thrld* | Video FIFO threshold in double words. This value is normally equivalent to line size divided by 8. Represents number of double words of Luminance data that are transferred per EDMA event. C threshold is half the Y threshold if Y threshold is even else will round to the nearest integer. When half line size or quarter line size threshold are provided (as required for HDTV and VESA modes), care should be taken to make sure that the Y and C line size are multiple of threshold. If not the driver will assume the threshold to be one line size. |
| *numFrmBufs* | Number of frame buffers that the driver allocates. If this is 0, then the driver operates in Normal mode. When this value is other than 0, then driver operates in Legacy mode. In both cases, minimum of 3 buffers are recommended for proper |

| | |
|---|---|
| | driver operation |
| *alignment* | Frame buffer alignment. This is used at the time of allocating buffer |
| *mergeFlds* | Field operation. Boolean:<br>*TRUE* = Fields are merged in interlaced operation<br>*FALSE* = Fields are separated in interlaced operation.<br>For progressive modes, this should be always *FALSE* |
| *segId* | Memory segment ID, used by driver at the time of allocating video frame buffer |
| *hEdma* | EDMA3 driver handle |

# 4 FVID API's

This chapter describes the functions, data structures, enumerations and macros for the VPORT driver module.

The following API functions are defined by the FVID module:

| | |
|---|---|
| ***FVID_create*** | Allocate buffers (optional) and initialize an FVID channel object |
| ***FVID_delete*** | De-allocate buffers (optional) and FVID channel object |
| ***FVID_alloc*** | Get a pointer for allocated buffer from driver to application (used only in Legacy Mode) |
| ***FVID_free*** | Relinquish a video buffer back to the driver (used only in Legacy Mode) |
| ***FVID_control*** | Send a control command to the mini-driver |
| ***FVID_exchange*** | Exchange an application-owned buffer for a driver-owned buffer |
| ***FVID_dequeue*** | Get a pointer for allocated buffer from driver to application (used only in Normal Mode) |
| ***FVID_queue*** | Relinquish a video buffer back to the driver (used only in Normal Mode) |
| ***FVID_allocBuffer*** | Allocate a frame buffer using the driver's memory allocation routines |
| ***FVID_freeBuffer*** | Free the buffer allocated via *FVID_allocBuffer()* |

## 4.1 Constants & Enumerations

### 4.1.1 Structure for Interlaced Frame

```
typedef struct FVID_IFrame_t
{
    Char* y1;           /* Character pointer for field 1 Y data */
    Char* cb1;          /* Character pointer for field 1 Cb data */
    Char* cr1;          /* Character pointer for field 1 Cr data */
    Char* y2;           /* Character pointer for field 2 Y data */
    Char* cb2;          /* Character pointer for field 2 Cb data */
    Char* cr2;          /* Character pointer for field 2 Cr data */
} FVID_IFrame;
```

This structure will be used in case of interlaced video mode.

If the driver is enabled for field separated mode (during initialization) then first 3 members will point to video data of field 1 and next 3 members will point to video data of field 2.

If the field separate option is disabled (fields merged), then only the first 3 member will point to video data of the whole frame and next 3 members are not used. (Logically same as *FVID_PFrame*)

### 4.1.2 Structure for Progressive Frame

```
typedef struct FVID_PFrame_t
{
   Char* y;                    /* Character pointer for frame Y data */
   Char* cb;                   /* Character pointer for frame Cb data */
   Char* cr;                   /* Character pointer for frame Cr data */
} FVID_PFrame;
```

### 4.1.3 Structure for Interlaced Raw Frame

```
typedef struct FVID_RawIFrame
{
   Char* buf1;          /* Character pointer for field 1 */
   Char* buf2;          /* Character pointer for field 2 */
} FVID_RawIFrame;
```

❑ This structure is not used in the current DM648 VPORT driver as it doesn't support interlaced RAW capture or RAW display. This is meant for future purpose.

### 4.1.4 Structure for Raw Frame

```
typedef struct FVID_RawPFrame
{
   Char* buf;           /* Character pointer for frame */
} FVID_RawPFrame;
```

### 4.1.5 Structure for FVID frame buffer descriptor

```
typedef struct _FVID_Frame
{

   QUE_Elem          queElement;      /* for queuing */

   union
   {
      FVID_IFrame       iFrm;         /* In/Out y/c frame buffer */
      FVID_PFrame       pFrm;         /* In/Out y/c frame buffer */
      FVID_RawIFrame    riFrm;        /*In/Out raw frame buffer */
      FVID_RawPFrame    rpFrm;        /* In/Out raw frame buffer */
      Ptr               frameBufferPtr; /* In/Out Raw Frame Buffer */
   } frame;

   Uint32            timeStamp;   /* Out (during each dequeue) Time Stamp
                                     */
   Uint32            pitch;       /* Out (during buffer alloc) Pitch
                                     parameters for given plane */
   Uint32            lines;       /* Out (during buffer alloc) Pointer to
                                     frambuffer for given plane */
   FVID_bitsPerPixel bpp;         /* Out (during buffer alloc) Bits per pixel
                                     support */
   FVID_colorFormat  frameFormat; /* In/Out Frame Color Format */

   Ptr               userParams;  /* In/Out Additional User Parameters per
                                     frame */
   Ptr               misc;        /* For future use */
} FVID_Frame;
```

This structure is the descriptor which consolidates the buffer pointers and other useful parameters.

The structure members *bpp* (bits per pixel), *frameFormat*, *pitch* and *lines* are updated during the time of buffer allocation.

The *pitch* member represents the Y-pitch of the FVID frame buffer. Since only YUV 422 planar format is supported, C- pitch will normally be half of Y-pitch, rounded to the nearest multiple of 8 due to alignment requirements. For example, if line size is 720 and if ½ scaling is enabled, then Y-pitch will be 360 and C-pitch will be 184 (not 180 as expected).

The structure members *timeStamp*, *userParams* and *queElement* are used in DM648 drivers and applications. They are used/updated for every frame exchange (queue/dequeue) operation.

The structure member ***misc*** is not used by the DM648 driver currently and is meant for future purpose.

DM648 VPORT driver only supports planar 422 formats. Planar format is used for all of the frame types. YUV 422 planar format is used for Y/C frame buffer (*iFrm* and *pFrm*). Note that Interleaved YUV 422 format is not supported. Frame types *riFrm* and *rpFrm* use raw planar format.

YUV 420 planar or interleaved frame format is not supported.

The YUV frame format is independent of the endianness since Y, Cb and Cr of each pixel are always 8 bit wide. For 16-bit raw frame format, each pixel data should be in little endian byte ordering.

This structure is backward compatible with DM642 applications. But recompilation of application with new API is required.

### 4.1.6 Enum for Color format

```
typedef enum _FVID_ColorFormat
{
   FVID_YCbCr422_INTERLEAVED = 0,
   FVID_YCbCr422_PLANAR,
   FVID_YCrCb422_INTERLEAVED,
   FVID_RGB_888_INTERLEAVED,
   FVID_RGB565_INTERLEAVED,
   FVID_DVD_MODE,
   FVID_CLUT_INDEXED,
   FVID_ATTRIBUTE,
   FVID_BAYER_PATTERN,
   FVID_RAW_FORMAT,
   FVID_COLORFORMAT_INVALID
} FVID_ColorFormat;
```

The ENUM string itself is self explanatory of the color format and **this is not used in DM648 driver** and these are for future use or for other drivers that share the FVID APIs. Only *FVID_YCbCr422_PLANAR* and *FVID_RAW_FORMAT* format are supported for. 16-bit RGB raw format is used for VESA displays.

### 4.1.7 Enum for Field Frame Modes

```
typedef enum _FVID_FieldFrame
{
   FVID_FIELD_MODE = 0,    /* Interlaced Mode */
   FVID_FRAME_MODE         /* Progressive  Mode */
} FVID_FieldFrame;
```

The ENUM string itself is self explanatory of the frame modes and **this is not used in DM648 driver** and these are for future use or for other drivers that share the FVID APIs.

### 4.1.8 Enum for Bits Per Pixel for different Module

```
typedef enum _FVID_bitsPerPixel
{
   FVID_BPP_BITS1   = 1,
   FVID_BPP_BITS2   = 2,
   FVID_BPP_BITS4   = 4,
   FVID_BPP_BITS8   = 8,
   FVID_BPP_BITS16 = 16,
   FVID_BPP_BITS24 = 24
} FVID_bitsPerPixel;
```

The ENUM string itself is self explanatory of the bits per pixel and **this is not used in DM648 driver** and these are for future use or for other drivers that share the FVID APIs. Only *FVID_BPP_BITS16* is supported.

### 4.1.9 Defines for IOM_Packet

```
#define FVID_BASE            IOM_USER
#define FVID_ALLOC           (FVID_BASE + 0)
#define FVID_FREE            (FVID_BASE + 1)
#define FVID_EXCHANGE        (FVID_BASE + 2)
#define FVID_QUEUE           (FVID_BASE + 3)
#define FVID_DEQUEUE         (FVID_BASE + 4)
#define FVID_ALLOC_BUFFER    (FVID_BASE + 5)
#define FVID_FREE_BUFFER     (FVID_BASE + 6)
```

These are command codes used in FVID to GIO API conversion macros.

## 4.2 API Definition

### 4.2.1 FVID_ create

**Syntax**

```
FVID_Handle FVID_create(
        String name,
        Int mode,
        Int *status,
        Ptr optArgs,
        FVID_Attrs *attrs
        );
```

**Parameters**

***name***

The *name* argument is the name specified for the device when it was created in the configuration or at runtime. It is used to find a matching name in the device table.
Note: strings are case sensitive.

For VPORT drivers the string is divided into 3 tokens separated by *'/'*.

- Video port driver or port instance
  This identifies the Video port driver or port instance. For capture drivers this will be typically *"VP0CAPTURE"*, *"VP1CAPTURE"* and so on. For display this will be typically *"VP0DISPLAY"*, *"VP1DISPLAY"* and so on. This string depends on the device registration string given in BIOS driver TCI file.

- Video port channel instance
  This identifies the channel to be opened in a port. Each Video capture port has two channels – *"A"* and *"B"*. Display ports support only single channel operation. Hence this token is not present for display drivers.

- EDC driver instance
  This identifies the External Device Control driver to be opened and linked to the video port driver instance. This token is typically more dependent on the EVM schematics and external encoders and decoders present in the EVM.

  ❑ If there is no requirement for EDC driver configuration for a VPORT driver instance, this token can be absent.

  In the present DM648 EVM, for capture this varies from *"0"* to *"7"* for TVP5154 SD decoders, *"0"* for TVP7000 HD decoder and for display this will be *"SAA7105"* or *"THS8200"* depending on the display mode of operation. Use *"SAA7105"* for SD display and use *"THS8200"* for HD and VESA display.

The following table shows the typical names for the current DM648 EVM

| String Name | Description |
|---|---|
| *"VP0CAPTURE/A/0"* | VPORT 0 Channel A capture – SD (TVP5154 #1) or HD (TVP7000) capture depends on EDC function pointer registered during bind device |
| *"VP0CAPTURE/B/1"* | VPORT 0 Channel B SD capture |
| *"VP1DISPLAY/SAA7105"* | VPORT 1 SD (NTSC/PAL) display |
| *"VP1DISPLAY/THS8200"* | VPORT 1 HD (720p, 1080i) and VESA display |
| *"VP2CAPTURE/A/2"* | VPORT 2 Channel A SD capture |
| *"VP2CAPTURE/B/3"* | VPORT 2 Channel B SD capture |
| *"VP3CAPTURE/A/4"* | VPORT 3 Channel A SD capture |
| *"VP3CAPTURE/B/5"* | VPORT 3 Channel B SD capture |
| *"VP4CAPTURE/A/6"* | VPORT 4 Channel A SD capture |
| *"VP4CAPTURE/B/7"* | VPORT 4 Channel B SD capture |

***mode***

The *mode* argument specifies the mode in which the device is to be opened. This may be *IOM_INPUT* or *IOM_OUTPUT*. *IOM_INPUT* mode is used for capture and *IOM_OUTPUT* mode is used for display.

***status***

The *status* argument is an output parameter that this function fills with a pointer to the status that was returned by the mini-driver.

***optArgs***

The *optArgs* parameter is a pointer that may be used to pass device or domain-specific arguments to the mini-driver. The contents at the specified address are interpreted by the mini-driver in a device-specific manner. The memory segment id for memory allocation is also passed via this parameter.

For DM648, optArgs will be pointer of type *VPORTCAP_Params* for capture driver (SD and HD) or *VPORTCAP_ParamsRaw* for raw capture driver or *VPORTDIS_Params* for display driver.

The VPORT driver doesn't assume any default value for this argument. This is because EDMA handle and Segment ID (used for frame buffer allocation) are passed to the driver only through this parameter. Hence VPORT driver will return error value if application passes NULL for this parameter.

***attrs***

The *attrs* parameter is a pointer to a structure of type *FVID_Attrs*. This is not supported and NULL should be passed.

**Return Value**

It returns the handle of type *FVID_Handle* on successful opening of a device. It returns NULL if the device could not be opened.

**Description**

An application calls *FVID_create* to create and initialize a video driver channel to the driver.

In Legacy mode this API is compatible to usage of DM642 FVID create API. The 'no. of buffers' ("*numFrmBufs"* member) in the channel parameters is the flag used to

identify this mode. The value must be greater than 0 for the driver to work in this mode.

In Legacy mode, this call will allocate frame buffers (Number of Buffers provided as parameter), initialize EDMA channels and configure video port registers. These frame buffers will be used later for FVID exchange and other APIs.

In Normal mode (where the *'numFrmBufs'* in the channel parameters is 0), the driver will not allocate frame buffers for FVID exchange and other APIs. Applications have to create buffers for this purpose. It is suggested that applications should use the APIs *FVID_allocBuffer* and *FVID_freeBuffer* provided with driver for frame buffer allocation purpose.

In both the modes, a minimum of 3 frame buffers should be used per driver instance for proper operation.

*FVID_create* returns a handle to the channel if it is successfully opened. This handle can then be used by subsequent FVID module calls to this channel.

For RAW capture operation, *optArgs* passed should be a pointer of *VPORTCAP_ParamsRaw* structure.

**Constraints**
This function can only be called after the device has been loaded and initialized.

**Example**
The example below shows creation of Capture Channel for VPORT

```
FVID_Handle chanHandle;

VPORTCAP_Params vCapParamsChan =
        CAP_PARAMS_CHAN_BT656_DEFAULT (NTSC);

chanHandle = FVID_create("/VP0CAPTURE/A/0", IOM_INPUT,
                         NULL, (Ptr)&vCapParamsChan, NULL);
if (NULL == chanHandle)
{
      printf(" Failed create capture channel \r\n");
      return;
}
```

### 4.2.2 FVID_delete

**Syntax**
*int FVID_delete(FVID_Handle fvidChan);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to FVID_create.

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred. This function is a wrapper above *GIO_delete()* function. Since *GIO_delete()* always returns success irrespective of VPORT driver return value, this function always returns *IOM_COMPLETED*.

**Description**
This function call will close the logical channel associated with *fvidChan* parameter. It will also free the buffers allocated by driver in Legacy mode i.e. DM642 compatibility mode. For Normal mode, it is the applications responsibility to free the already allocated buffers before channel deletion. Kindly note that, if capture/display operation is started, then *VPORT_CMD_STOP* should be called before calling *FVID_delete.*

EDC driver associated with the channel is also closed in this function call.

**Constraints**
This function can only be called after the device has been loaded, initialized and created.

**Example**
The example below shows creation and deletion of Capture Channel for VPORT

```
FVID_Handle chanHandle;

VPORTCAP_Params vCapParamsChan =
        CAP_PARAMS_CHAN_BT656_DEFAULT (NTSC);

chanHandle = FVID_create("/VP0CAPTURE/A/0", IOM_INPUT,
                         NULL, (Ptr)&vCapParamsChan, NULL);
if (NULL == chanHandle)
{
      printf(" Failed create capture channel \r\n");
      return;
}

FVID_delete(chanHandle);
```

### 4.2.3 FVID_alloc

**Syntax**
*int FVID_alloc(FVID_Handle fvidChan, Ptr bufp);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*

***bufp***
The *bufp* argument is an out parameter that this function fills with a pointer to the structure of type *FVID_Frame* that is owned by the video port driver.

**Return Value**
*FVID_alloc* returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned. If there is no buffer available with driver to return to application, this function will be blocked.

**Description**
This API is valid only in Legacy mode (DM642 compatibility mode). In the Normal mode *FVID_dequeue* should be used instead of this API and **the DM648 driver will return error for this API call.**

An application will call *FVID_alloc* to request the video device driver to relinquish ownership of a frame buffer. This API function will result in an *mdSubmit* call being made to the mini-driver.

For display operation, the driver will return an empty frame buffer which the application can use to fill the next frame data to be displayed. For capture operation, the driver will return the most recently captured frame buffer which can be used by the application for further processing.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. Cache coherency of the frame buffer should be taken care by the application.

**Example**
```
FVID_Handle  chanHandle;
FVID_Frame   *capBuffer;

/* channel creation should be done here */

/* get a buffer from the device */
status = FVID_alloc(chanHandle, &capBuffer);
```

### 4.2.4 FVID_free

**Syntax**
*int FVID_free (FVID_Handle fvidChan, Ptr bufp);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*

***bufp***
The *bufp* argument is a pointer to the structure of type *FVID_Frame* that was previously allocated by the device driver and needs to be relinquished by the application.

**Return Value**
*FVID_free* returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned.

**Description**
This API is valid only in Legacy mode (DM642 compatibility mode). In the Normal mode *FVID_queue* should be used instead of this API and **the DM648 driver will return error for this API call.**

An application calls *FVID_free* to relinquish a video buffer back to the video device driver. This API function will result in an *mdSubmit* call being made to the mini-driver.

For display operation, the application gives a filled frame buffer that needs to be displayed next. For capture operation, the application gives an empty buffer to the driver for capturing the next frame data.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. Cache coherency of the frame buffer should be taken care by the application.

The pointer that is passed as an argument to this call must point to a video buffer of type *FVID_Frame*. This pointer should point to buffer already provided by the driver through a call to *FVID_alloc* or *FVID_exchange*

**Example**
```
FVID_Handle  chanHandle;
FVID_Frame   *capBuffer;

/* channel creation & alloc should be done here */

status = FVID_free(chanHandle, &capBuffer);
```

### 4.2.5 FVID_control

**Syntax**
*int FVID_control (fvidChan, cmd, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
The *cmd* argument specifies the control command
***args***
The *args* argument is a pointer to the argument or structure of arguments that are specific to the command being passed.

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
An application calls *FVID_control* to send device-specific control commands to the mini-driver.

Below are the supported control commands by DM648 VPORT driver. The following sections explain the commands in detail.

- *VPORT_CMD_CONFIG_CHAN*
  Reconfigures capture or display channel. This command can be used to change channel configuration at runtime. **Note:** Driver mode (Legacy or Normal) cannot be changed at runtime

- *VPORT_CMD_START*
  Start display/capture operation

- *VPORT_CMD_STOP*
  Stop display/capture operation

- *VPORT_CMD_SET_VINTCB*
  Configures VPORT interrupt settings and register a callback for sync interrupts and error conditions

- *VPORT_CMD_COVR_RECOVER*
  On Capture overrun this will reset VPORT for further operation

- *VPORT_CMD_DUND_RECOVER*
  On Display under run, this will reset VPORT for further operation

- *VPORT_CMD_GET_NUM_IORQST_PENDING*
  Gets the number of pending request at driver level

- *VPORT_CMD_GET_PARAMS*
  Get the current channel configuration parameters of driver

- *VPORTCAP_CMD_SET_LINE_INT*
  Register call-back function to get interrupts after specified number of lines is captured by VPORT.

- *VPORTCAP_CMD_GET_NUMLINES_CAPTURED*
  Returns the current video frame pointer and number of lines captured in the current video frame

- Default IOCTL
  Used to configure the external encoders and decoders. Interface will depend on the encoder/decoder drivers

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.

This function is not re-entrant for a channel.

**Example**
```
FVID_Handle chanHandle;
/* channel creation should be done here */

/* start capture from video port */
status = FVID_control(chanHandle, VPORT_CMD_START, NULL);
if (IOM_COMPLETED != status)
{
      printf(" Failed start capture channel \r\n");
      return;
}
```

Below section describes different IOCTL commands supported by *FVID_control()*:

#### 4.2.5.1 VPORT_CMD_CONFIG_CHAN

**Syntax**
*int FVID_control (fvidChan, VPORT_CMD_CONFIG_CHAN, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.

***cmd***
*VPORT_CMD_CONFIG_CHAN* control command

***args***
The *args* argument is a pointer to structure containing the new configuration and is of type *VPORTCAP_Params* for capture driver or *VPORTCAP_ParamsRaw* for RAW capture driver or *VPORTDIS_Params* for display driver.

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
This function call is used to change channel configuration at runtime. Application can call this function to change video port configuration or number of buffers to be used in Legacy mode. The argument to this command is similar to *optargs* parameter in *FVID_create*.

In LEGACY MODE, if application calls this function when channel is started then driver will start the channel again after reconfiguration. In Normal mode, if application calls this function when channel is started then driver will reconfigure channel but will not start channel. Application has to queue buffers before starting channel again.

In LEGACY MODE, this call will free memory for all the buffers and allocate new buffers as part of channel reconfiguration. In NORMAL MODE, it is application's responsibility to free memory for all the buffers before reconfiguring channel.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.
Application can't change driver mode (Legacy mode or Normal mode) using this function. Application shall not change EDMA handle in argument to this function call. Driver will use the EDMA handle passed at time of *FVID_create* only.

This function is not re-entrant for a channel.

**Example**

```
FVID_Handle chanHandle;
VPORTCAP_Params vCapParamsChan =
        CAP_PARAMS_CHAN_BT656_DEFAULT (NTSC);

/* channel creation should be done here */

/* configure capture channel */
status = FVID_control(chanHandle, VPORT_CMD_CONFIG_CHAN,
                        &vCapParamsChan);
if (IOM_COMPLETED != status)
{
      printf(" Failed to configure capture channel \r\n");
      return;
}
```

#### 4.2.5.2 VPORT_CMD_START

**Syntax**
*int FVID_control (fvidChan, VPORT_CMD_START, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
*VPORT_CMD_START* control command
***args***
None

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
This function call is used to start capture or display operation.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.
In NORMAL MODE, this function can be called only after minimum required buffers are queued up.

This function is not re-entrant for a channel.

**Example**
```
FVID_Handle chanHandle;

/* channel creation should be done here */

/* configure capture channel */
status = FVID_control(chanHandle, VPORT_CMD_START, NULL);

if (IOM_COMPLETED != status)
{
      printf(" Failed to start  capture channel \r\n");
      return;
}
```

#### 4.2.5.3 VPORT_CMD_STOP

**Syntax**
*int FVID_control (fvidChan, VPORT_CMD_STOP, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
*VPORT_CMD_STOP* control command
***args***
None

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
This function call is used to stop capture or display operation.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.
This function can be called only after capture or display operation has started.

This function is not re-entrant for a channel.

**Example**
```
FVID_Handle chanHandle;

/* channel creation should be done here */

/* stop capture channel */
status = FVID_control(chanHandle, VPORT_CMD_STOP, NULL);
if (IOM_COMPLETED != status)
{
      printf(" Failed to stop  capture channel \r\n");
      return;
}
```

#### 4.2.5.4 VPORT_CMD_SET_VINTCB

**Syntax**
*int FVID_control (fvidChan, VPORT_CMD_SET_VINTCB, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.

***cmd***
*VPORT_CMD_SET_VINTCB* control command

***args***
The *args* argument is a pointer to structure of type *VPORT_VIntCbParams*.

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
This function call is used to configure VPORT interrupt and register a callback for the same. VPORT provides various interrupts for different sync scenarios and error conditions.

Application can enable multiple interrupts simultaneously, but application can get a common callback for all the interrupts registered per channel.

Register mask values for different interrupt are defined in *vport.h*. By default the driver handles display under run and capture over run errors. Application doesn't have to do any recovery if these errors occur.

❑ If the application registers both *VPORT_CMD_SET_VINTCB* and *VPORTCAP_CMD_SET_LINE_INT* IOCTLs with the driver, then the driver will take the lastly passed vertical line number for VPORT interrupt generation. But the application still receives two callbacks as these two callbacks are independent of each other except for the vertical line number.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.

❑Note: Vertical line interrupts are not supported for RAW capture.

⚐ This function is not re-entrant for a channel.

**Example**

```
FVID_Handle chanHandle;
VPORT_VIntCbParams setcbparams;

/* channel creation should be done here */

/* pointer to the video port interrupt call-back function */
setcbparams.vIntCbFxn = vport_intCallbackFunc;

/* vertical field 1 and field 2 line interrupts enabled */
setcbparams.vIntMask = (VPORT_INT_VINT2 | VPORT_INT_VINT1);

/* line number where vertical interrupt occurs */
setcbparams.vIntLine = 50;

/* configure vport interrupt callback */
status = FVID_control(chanHandle, VPORT_CMD_SET_VINTCB, &setcbparams);

if (IOM_COMPLETED != status)
{
      printf(" Failed to configure vport interrupt callback \r\n");
      return;
}
```

#### 4.2.5.5 VPORT_CMD_COVR_RECOVER

**Syntax**
*int FVID_control (fvidChan VPORT_CMD_COVR_RECOVER, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
*VPORT_CMD_COVR_RECOVER* control command
***args***
NULL

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
On Capture over run error, this function call will reset Video Port for further operation. If capture over run callback is enabled using *VPORT_CMD_SET_VINTCB* command and if driver reports capture over run error to application then application can call *VPORT_CMD_COVR_RECOVER* to reconfigure EDMA transfer and enable capture operation again. By default the driver handles capture over run error. Application doesn't have to do any recovery if this error occurs.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.
In NORMAL MODE, this function can be called only after minimum required buffers are queued up.

⚐ This function is not re-entrant for a channel.

**Example**
```
FVID_Handle chanHandle;

/* Below call is made if driver notifies capture over run error to application */
/* call capture over run recovery function */
status = FVID_control(chanHandle, VPORT_CMD_COVR_RECOVER, NULL);

if (IOM_COMPLETED != status)
{
      printf(" Failed in capture over run recovery \r\n");
      return;
}
```

#### 4.2.5.6 VPORT_CMD_DUND_RECOVER

**Syntax**
*int FVID_control (fvidChan VPORT_CMD_DUND_RECOVER, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
*VPORT_CMD_DUND_RECOVER* control command
***args***
NULL

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
On Display under run error, this function call will reset Video Port for further operation. If display under run callback is enabled using *VPORT_CMD_SET_VINTCB* command and if driver reports display under run error to application then application can call *VPORT_CMD_DUND_RECOVER* to reconfigure EDMA transfer and enable display operation again. By default the driver handles display under run error. Application doesn't have to do any recovery if this error occurs.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.
In NORMAL MODE, this function can be called only after minimum required buffers are queued up.

⚐ This function is not re-entrant for a channel.

**Example**
```
FVID_Handle chanHandle;

/* Below call is made if driver notifies display under run error to application */
/* call display under run recovery function */
status = FVID_control(chanHandle, VPORT_CMD_DUND_RECOVER, NULL);

if (IOM_COMPLETED != status)
{
      printf(" Failed in display under run recovery \r\n");
      return;
}
```

#### 4.2.5.7 VPORT_CMD_GET_NUM_IORQST_PENDING

**Syntax**
*int FVID_control (fvidChan VPORT_CMD_GET_NUM_IORQST_PENDING, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
*VPORT_CMD_GET_NUM_IORQST_PENDING* control command
***args***
Pointer to Integer

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
This function call will get number of pending requests at driver level. It will provide number of requests yet to be served by driver.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.
In NORMAL MODE, this function can be called only after minimum required buffers are queued up.

This function is not re-entrant for a channel.

**Example**
```
FVID_Handle chanHandle;
Int numPendingReq;

/* channel creation should be done here */

/* call to get number of pending requests */
status = FVID_control(chanHandle, VPORT_CMD_GET_NUM_IORQST_PENDING,
                      &numPendingReq);
if (IOM_COMPLETED != status)
{
      printf(" Failed in getting pending requests \r\n");
      return;
}
```

#### 4.2.5.8 VPORT_CMD_GET_PARAMS

**Syntax**
*int FVID_control (fvidChan VPORT_CMD_GET_PARAMS, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
*VPORT_CMD_GET_PARAMS* control command
***args***
Pointer to structure of type *VPORT_CurrParams*

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
This function will provide current channel configuration parameters of capture or display driver.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.

⚐ This function is not re-entrant for a channel.

**Example**
```
FVID_Handle chanHandle;
VPORT_CurrParams currParams;

/* channel creation should be done here */

/* call to get number of pending requests */
status = FVID_control(chanHandle, VPORT_CMD_GET_PARAMS,
                      &currParams);
if (IOM_COMPLETED != status)
{
      printf(" Failed in getting current params \r\n");
      return;
}
```

#### 4.2.5.9 VPORTCAP_CMD_SET_LINE_INT

**Syntax**
*int FVID_control (fvidChan VPORTCAP_CMD_SET_LINE_INT, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
*VPORTCAP_CMD_SET_LINE_INT* control command
***args***
Pointer to structure of type *VPORTCAP_LineIntParam*

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
This command is used to enable line interrupt of video port and its callback to application. This is similar to vertical interrupt enabled using *VPORT_CMD_SET_VINTCB,* but its callback function will provide the current frame buffer pointer to application. This function applies to capture driver only.

❑ If the application registers both *VPORT_CMD_SET_VINTCB* and *VPORTCAP_CMD_SET_LINE_INT* IOCTLs with the driver, then the driver will take the lastly passed vertical line number for VPORT interrupt generation. But the application still receives two callbacks as these two callbacks are independent of each other except for the vertical line number.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.

❑ Note: This IOCTL is not valid for RAW capture.

⚐ In the current driver implementation, this interrupt is based on Vertical Line interrupts provided by VPORT. Hence when this interrupt occurs it is not necessary that the EDMA has actually transferred that much number of lines due to latency between VPORT FIFO and EDMA transfer to memory.

For the same reason even the frame buffer pointer could be invalid (point to previous frame) if vertical interrupts are set near to the start of the frame.

⚐ This function is not re-entrant for a channel.

**Example**

```
FVID_Handle chanHandle;
VPORTCAP_LineIntParam vintparams;


/* channel creation should be done here */


/* pointer to the video port interrupt call-back function */
vintparams.callBackFxn = (VPORTCAP_LinesCapturedCB) lineIntCallbackFunc;

vintparams.cbArg = 6;       /* callback argument */

/* line number where vertical interrupt occurs */
vintparams.vIntFld1 = 20;
vintparams.vIntFld2 = 70;


/* configure vport interrupt callback */
status = FVID_control(chanHandle, VPORTCAP_CMD_SET_LINE_INT,
                      &vintparams);

if (IOM_COMPLETED != status)
{
      printf(" Failed to configure vertical line interrupt callback \r\n");
      return;
}
```

#### 4.2.5.10 VPORTCAP_CMD_GET_NUMLINES_CAPTURED

**Syntax**
*int FVID_control (fvidChan, VPORTCAP_CMD_GET_NUMLINES_CAPTURED, args);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*.
***cmd***
*VPORTCAP_CMD_GET_NUMLINES_CAPTURED* control command
***args***
Pointer to structure of type *VPORTCAP_LinesCapturedInfo*

**Return Value**
*IOM_COMPLETED* on success, or negative value if an error occurred

**Description**
This function will provide number of lines captured by EDMA for current frame buffer. It will also provide current frame buffer pointer in which EDMA is transferring captured data from Video port FIFO. This function applies to capture driver only.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *FVID_create*.
This function can be called only after capture operation has started.

- This IOCTL reads the current EDMA channel parameters and determine the number of lines captured by EDMA. Since EDMA reload of channel parameter is a hardware event, correctness of the lines captured at the end of the frame cannot be ensured. It is left to the application to determine whether the lines captured in the current frame is proper especially at the end of the frame.

- This function is not re-entrant for a channel.

**Example**

```
FVID_Handle chanHandle;
VPORTCAP_LinesCapturedInfo capLineInfo;

/* channel creation should be done here */

/* call to get number of pending requests */
status =
FVID_control(chanHandle, VPORTCAP_CMD_GET_NUMLINES_CAPTURED,
                &capLineInfo);
if (IOM_COMPLETED != status)
{
      printf(" Failed in getting number of lines captured by EDMA \r\n");
      return;
}
```

### 4.2.6 FVID_exchange

**Syntax**
*int FVID_exchange(FVID_Handle fvidChan, Ptr bufp);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*

***bufp***
The *bufp* argument is an in/out parameter that points to the application-owned buffer that is to be relinquished back to the driver. After the call returns successfully, this function fills bufp with a pointer to the structure of type *FVID_Frame* that was exchanged by the device driver.

**Return Value**
*FVID_exchange* returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned.

**Description**
An application calls *FVID_exchange* to relinquish a video buffer back to the video device driver. After the call returns successfully, this function fills bufp with a pointer to the structure of type *FVID_Frame* that was exchanged by the device driver. This API function will result in an *mdSubmit* call being made to the mini-driver.

This operation is similar to calling *FVID_free* (*FVID_queue*) and *FVID_alloc* (*FVID_dequeue*) one after the other. Refer corresponding API description for details.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. Cache coherency of the frame buffer should be taken care by the application.

**Example**
```
FVID_Handle             chanHandle;
FVID_Frame*             capBuffer;
/* channel creation & alloc should be done here */

status = FVID_exchange(chanHandle, &capBuffer);
```

### 4.2.7 FVID_dequeue

**Syntax**
*int FVID_dequeue (FVID_Handle fvidChan, Ptr bufp);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*

***bufp***
The *bufp* argument is an out parameter that this function fills with a pointer to the structure of type *FVID_Frame* that was allocated by the device driver.

**Return Value**
*FVID_dequeue* returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned. If there is no buffer available with driver to return to application, this function will be blocked. But if application calls *FVID_dequeue* after calling *VPORT_CMD_STOP* and if there is no buffer available with driver to return to application, then *IOM_ENOPACKETS* code will be returned.

**Description**
This API is NOT supported for Legacy mode (DM642 compatibility mode) and is supported only in the Normal mode.

An application will call *FVID_dequeue* to request the video device driver to give ownership of a data buffer. This API function will result in an *mdSubmit* call being made to the mini-driver.

For display operation, the driver will return an empty frame buffer which the application can use to fill the next frame data to be displayed. For capture operation, the driver will return the most recently captured frame buffer which can be used by the application for further processing.

After the channel is stopped, this function is used to get all the buffers owned by the driver to free it by calling *FVID_freeBuffer* API.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. Cache coherency of the frame buffer should be taken care by the application.

And this function should be called only after queuing minimum number of buffers to the drivers.

**Example**
```
FVID_Handle  chanHandle;
FVID_Frame   *capBuffer;

/* channel creation should be done here */

/* get a buffer from the device */
status = FVID_dequeue(chanHandle, &capBuffer);
```

### 4.2.8 FVID_queue

**Syntax**
*int FVID_queue(FVID_Handle fvidChan, Ptr bufp);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*

***bufp***
The *bufp* argument is a pointer to the structure of type *FVID_Frame* that was previously allocated by the device driver and is not to be relinquished.

**Return Value**
*FVID_queue* returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned.

**Description**
This API is NOT supported for Legacy mode (DM642 compatibility mode) and is supported only in the Normal mode.

An application calls *FVID_queue* to submit a video buffer to the video device driver. This API function will result in an *mdSubmit* call being made to the mini-driver.

For display operation, the application gives a filled frame buffer that needs to be displayed next. For capture operation, the application gives an empty buffer to the driver for capturing the next frame data.

Before the channel is started, this function is used to queue the required number of buffers allocated by calling *FVID_allocBuffer* API.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. Cache coherency of the frame buffer should be taken care by the application.

The pointer that is passed as an argument to this call must point to a video buffer of type *FVID_Frame*. This pointer must point to either the buffer newly allocated or the buffer already provided by the driver through a call to *FVID_dequeue* or *FVID_exchange or FVID_allocBuffer* calls.

**Example**
```
FVID_Handle  chanHandle;
FVID_Frame   *capBuffer;

/* channel creation should be done here */

status = FVID_queue (chanHandle, &capBuffer);
```

### 4.2.9 FVID_allocBuffer

**Syntax**
*int FVID_allocBuffer (FVID_Handle fvidChan, Ptr bufp);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to FVID_create

***bufp***
The bufp argument is an out parameter which will contain pointer to the allocated frame buffer from the segment ID provided in create parameter.

**Return Value**
*FVID_allocBuffer* returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned.

**Description**
This API is not supported in Legacy mode (DM642 compatibility mode) and is supported only in the Normal mode.

An application will call *FVID_allocBuffer* to request the video device driver to allocate one data buffer. This function allocates memory for one frame buffer and one structure variable of type *FVID_Frame*. This function fills buffer pointer in *FVID_Frame* structure variable and assigns its pointer to the structure pointer of type *FVID_Frame* passed as an argument. This API function will result in an *mdControl* call being made to the mini-driver. The segment ID passed to the driver during *FVID_create* will be used for allocation.

It is the responsibility of the application to dequeue the buffer from driver and free it before the channel is deleted.

**Constraints**
This function can only be called after the device has been loaded, initialized and created.

**Example**
```
FVID_Handle  chanHandle;
FVID_Frame   *capBuffer = NULL;

/* channel creation should be done here */

/* allocate a buffer from the device and get its pointer*/
status = FVID_allocBuffer (chanHandle, &capBuffer);
```

### 4.2.10FVID_freeBuffer

**Syntax**
*int FVID_freeBuffer (FVID_Handle fvidChan, Ptr bufp);*

**Parameters**
***fvidChan***
Handle of the video driver channel that was created with a call to *FVID_create*

***bufp***
The *bufp* argument will contain pointer to the frame buffer that is to be released.

**Return Value**
*FVID_freeBuffer* returns *IOM_COMPLETED* when it completes successfully. If an error occurs, a negative value will be returned.

**Description**
This API is not supported in Legacy mode (DM642 compatibility mode) and is supported in the Normal mode.

An application will call *FVID_freeBuffer* to request the video device driver to free memory of one data buffer. Pointer to this data buffer will be passed as an argument to *FVID_freeBuffer*. This API call will free memory of one data buffer and one *FVID_Frame* structure variable. This API function will result in an *mdControl* call being made to the mini-driver.

**Constraints**
This function can only be called after the device has been loaded, initialized and created. The pointer that is passed as an argument to this call must point to a video buffer of type *FVID_Frame*. This pointer must point to buffer already allocated by the driver through a call to *FVID_allocBuffer*.

**Example**
```
FVID_Handle  chanHandle;
FVID_Frame   *capBuffer;

/* channel creation and allocBuffer should be done here */

status = FVID_freeBuffer (chanHandle, &capBuffer);
```

### 4.2.11Using FVID API's

The following is a simplified example of an application that is capturing data from a video source and displaying the data to a display device.

#### *4.2.11.1 Examples for Legacy mode (DM642 compatibility mode) API usage*

```
#include <std.h>
#include <fvid.h>

main(void)
{
/* DSP/BIOS scheduler starts at the termination of main() */
}

/* Video processing task */
void tskVLoopback()
{
        /* capture/display channel objects */
        FVID_Handle capChan, disChan;

        /* capture/display frame buffers */
        FVID_Frame *capFrameBuf, *disFrameBuf;

        /* allocate buffers at the time of device create itself */
        capParams.numFrmBufs = 3;
        disParams.numFrmBufs = 3;

        /* create and initialize the FVID channel objects */
        capChan = FVID_create("/VP0CAPTURE/A/0", IOM_INPUT, NULL,
                        (Ptr)&capParams, NULL);
        disChan = FVID_create("/VP1DISPLAY/SAA7105", IOM_OUTPUT, NULL,
                        (Ptr)&disParams, NULL);

        /* start capture & display operation */
        FVID_control(capChan, VPORT_CMD_START, NULL);
        FVID_control(disChan, VPORT_CMD_START, NULL);

        /* Let application have ownership of the first set of buffers */
        FVID_alloc(capChan, &capFrameBuf);
        FVID_alloc(disChan, &disFrameBuf);

        while (j < NUM_FRAMES)
        {
                /* copy captured frame data to the display frame buffer */
                FrameDataCopy(capFrameBuf, disFrameBuf);
                FVID_exchange(capChan, &capFrameBuf);
                FVID_exchange(disChan, &disFrameBuf);
                j++;
        }
```

```
        /* stop capture & display operation */
        FVID_control(capChan, VPORT_CMD_STOP, NULL);
        FVID_control(disChan, VPORT_CMD_STOP, NULL);

        /* Delete channels and free buffers */
        FVID_delete(capChan);
        FVID_delete(disChan);
}
```

#### 4.2.11.2 Example for Normal mode API usage

```
#include <std.h>
#include <fvid.h>
#include <tsk.h>

#define NUM_BUFFERS             (3)

main(void)
{
/* DSP/BIOS scheduler starts at the termination of main() */
}

/* Video processing task */
void tskVLoopback(void)
{
        /* capture/display channel objects */
        FVID_Handle capChan, disChan;

        /* capture/display frame buffers */
        FVID_Frame *capFrameBuf, *disFrameBuf;

        /* do not allocate buffers at the time of device create */
        capParams.numFrmBufs = 0;
        disParams.numFrmBufs = 0;

        /* create and initialize the FVID channel objects */
        capChan = FVID_create("/VP0CAPTURE/A/0", IOM_INPUT, NULL,
                        (Ptr)&capParams, NULL);
        disChan = FVID_create("/VP1DISPLAY/SAA7105", IOM_OUTPUT, NULL,
                        (Ptr)&disParams, NULL);


        for (i=0; i < NUM_BUFFERS; i++)
        {
                /* Allocate buffers */
                FVID_allocBuffer(capChan, &capFrameBuf);
                FVID_allocBuffer(disChan, &disFrameBuf);

                /* Queue buffers to driver */
                FVID_queue(capChan, &capFrameBuf);
                FVID_queue(disChan, &disFrameBuf);
        }
```

```
        /* start capture & display operation */
        FVID_control(capChan, VPORT_CMD_START, NULL);
        FVID_control(disChan, VPORT_CMD_START, NULL);

        /* Let application have ownership of the first set of buffers */
        FVID_dequeue(capChan, &capFrameBuf);
        FVID_dequeue(disChan, &disFrameBuf);

        while (j < NUM_FRAMES)
        {
                /* copy captured frame data to the display frame buffer */
                FrameDataCopy(capFrameBuf, disFrameBuf);

                FVID_exchange(capChan, &capFrameBuf);
                FVID_exchange(disChan, &disFrameBuf);
                j++;
        }

        /* stop capture & display operation */
        FVID_control(capChan, VPORT_CMD_STOP, NULL);
        FVID_control(disChan, VPORT_CMD_STOP, NULL);

        /* Free the application owned buffer */
        FVID_freeBuffer(capChan, &capFrameBuf);
        FVID_freeBuffer(disChan, &disFrameBuf);

        for (i=0; i < (NUM_BUFFERS - 1); i++)
        {
                /* Dequeue buffers from driver */
                FVID_dequeue(capChan, &capFrameBuf);
                FVID_dequeue(disChan, &disFrameBuf);

                /* Free buffers */
                FVID_freeBuffer(capChan, &capFrameBuf);
                FVID_freeBuffer(disChan, &disFrameBuf);
        }

        /* Delete channels and free buffers*/
        FVID_delete(capChan);
        FVID_delete(disChan);
}
```

# 5 External Device Control (EDC)

This chapter describes in detail about External Device Control (EDC) mechanism of VPORT driver - EVM or hardware dependent components that are not built inside VPORT module and VPORT has dependency on such peripherals. DM648 video port driver configures external video decoders and encoders using I2C interface to capture or display video.

This chapter describes the functions, data structures and enumerations for the EDC module.

Most of the functionality and features supported by the EDC driver depends on the DM648 EVM schematics and VPORT support. Features which are not supported by the current DM648 EVM and VPORT are mentioned as NOT SUPPORTED in the appropriate places. The options which are not supported are given only for future purpose.

## 5.1 Interface between VPORT and EDC Driver

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

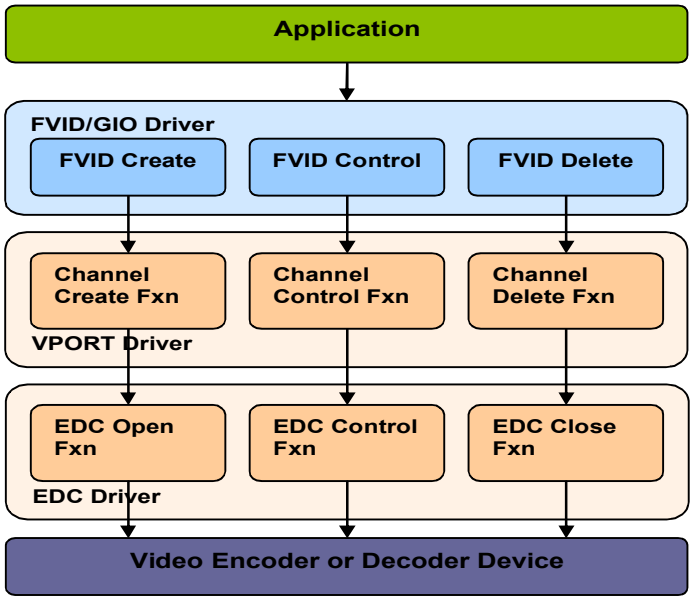


**Figure 3.** Interface between VPORT and EDC Driver

The EDC driver is associated with each instance of the VPORT driver through the *edcTbl[2]* member (of type *EDC_Fxns*) of *VPORT_PortParams*. This is done during VPORT driver registration through BIOS TCI file. Each VPORT can be associated with two EDC drivers, one for each of the two channel of VPORT. Since dual channel is not supported for display operation, only one EDC driver is associated with a display port.

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

```
typedef struct EDC_Fxns {
    EDC_Handle (*open)(String name, Arg optArg);
    Int (*close)(Ptr devHandle);
    Int (*ctrl)(Ptr devHandle, Uns cmd, Arg arg);
} EDC_Fxns;
```

The DM648 EVM has the following external encoders and decoders. The details of each driver interface are explained in the following section.

- Two TVP5154 Decoders
  Used to capture 8 NTSC/PAL resolutions

- One SAA7105 Encoder
  Used to display NTSC/PAL resolutions

- One THS8200 Encoder
  Used to display HDTV 720P, HDTV 1080I and VESA resolutions

- One TVP7000 Decoder
  Used to capture HDTV 720P and HDTV 1080I resolutions

## 5.2 TVP5154 Decoder

TVP5154 is a 4-channel, low-power, NTSC/PAL/SECAM video decoder. Each channel of the TVP5154 decoder converts NTSC, PAL, or SECAM video signals to 8-bit ITU-R BT.656 format.

The current DM648 EVM contains 2 TVP5154 decoders capable of capturing 8 (2 x 4) SD video channels.

TVP5154 input and output interface details are given below:

**Analog Input Interface:**

- Composite video – Selectable Channel A or B (Channel B not supported)
- S-video (Not supported)

**Digital Output Interface:**

- 8-bit BT656, With Embedded Sync
- 8-bit BT656, With External Sync (Not supported)

TVP5154 video decoder is an independent interface which is being configured from the VPORT driver. TVP5154 is I2C slave device. TVP5154 driver configures TVP5154 device using I2C interface.

### 5.2.1 Interface Functions

TVP5154 exports its function table pointer through *TVP5154_Fxns* global variable as defined below:

```
EDC_Fxns TVP5154_Fxns = {
    TVP5154_open,
    TVP5154_close,
    TVP5154_ctrl
};
```

To use TVP5154, application shall pass this function table pointer as part of device parameters (*edcTbl[2]* of *VPORT_PortParams*) during driver registration using BIOS TCI file. This will associate the EDC driver instance with the corresponding VPORT driver instance.

As shown in *Figure 3*, when application calls *FVID_create*, VPORT driver will internally call *TVP5154_open* function. This will power on TVP5154 device and initialize I2C driver for serial communication. One of strings *"0"* to *"7"* should be passed as argument to *TVP5154_open* function to open the corresponding decoder channel.

To configure TVP5154, application has to call *FVID_control* function with *VPORT_CMD_EDC_BASE + TVP5154 IOCTL* (as shown in below table) as command. This will internally call *TVP5154_ctrl* function. Once the VPORT driver deletes the channel, it will delete the TVP5154 driver instance and close the I2C driver as well.

TVP5154 driver provides support for different IOCTL commands as shown below. Application can call *FVID_control* with one of below specified IOCTL command and corresponding argument to configure TVP5154.

| TVP5154 IOCTL Command | Argument | Description |
|---|---|---|
| *EDC_CONFIG* | Pointer to structure variable of type *TVP5154_ConfParams* | Configures TVP5154 decoder parameters associated with this channel |
| *EDC_RESET* | NULL | Resets TVP5154 decoder channel |
| *TVP5154_POWERDOWN* | NULL | Power down the specified decoder channel |
| *TVP5154_POWERUP* | NULL | Power up the specified decoder channel |
| *TVP5154_SET_ANALOG_CHAN* | Pointer to structure variable of type *TVP5154_AnalogChan* | Selects analog input channel for TVP5154 decoder |

### 5.2.2 Data Structures

This section describes TVP5154 data structures exposed to the application.

❖ **TVP5154_ConfParams**

*"tvp5154.h"* file contains *TVP5154_ConfParams* data structure that is passed as an argument while calling *EDC_CONFIG* IOCTL for TVP5154 from the application. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *mode* | Type of enum *TVP5154_Mode*, indicates analog input standard for TVP5154 |
| *aFmt* | Type of enum *TVP5154_AnalogFormat*, indicates analog input format for TVP5154 |
| *enableBT656Sync* | Boolean to select embedded or external sync for digital output to VPORT<br>TRUE = Use embedded sync<br>FALSE = Use external sync (Not supported) |

### 5.2.3 Enumerations

❖ **TVP5154_AnalogFormat**

*"tvp5154.h"* file contains *TVP5154_AnalogFormat* enum that is passed while calling *EDC_CONFIG* IOCTL for TVP5154 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *TVP5154_AFMT_COMPOSITE_A* | Analog channel A selection for composite video input |
| *TVP5154_AFMT_COMPOSITE_B* | Analog channel B selection for composite video input (Not supported) |
| *TVP5154_AFMT_SVIDEO* | S-video selection (Not supported) |

❖ **TVP5154_AnalogChan**

*"tvp5154.h"* file contains *TVP5154_AnalogChan* enum that is passed while calling *TVP5154_SET_ANALOG_CHAN* IOCTL from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *TVP5154_AFMT_CHANNEL_A* | Analog channel A selection for composite video input |
| *TVP5154_AFMT_CHANNEL_B* | Analog channel B selection for composite video input (Not supported) |

❖ **TVP5154_Mode**

*"tvp5154.h"* file contains *TVP5154_Mode* enum that is passed while calling *EDC_CONFIG IOCTL* for TVP5154 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *TVP5154_MODE_NTSC* | Analog input standard is NTSC |
| *TVP5154_MODE_PAL* | Analog input standard is PAL |
| *TVP5154_MODE_SECAM* | Not supported |

## 5.3 SAA7105 Encoder

SAA7105 is an advanced next-generation video encoder which converts PC graphics data at maximum 1280x1024 resolutions (optionally 1920x1080 interlaced) to PAL (50 Hz) or NTSC (60 Hz) video signals. A programmable scalar and anti-flicker filter (maximum 5 lines) ensures properly sized and flicker-free TV display as CVBS or S-video output.

In DM648 EVM, SAA7105 encoder is connected to the VPORT through THS8200 BT.656 bypass port. SAA7105 encoder is used for NTSC/PAL SD resolution displays and THS8200 is used for EDTV, HDTV and VESA resolution displays.

SAA7105 input and output interface details are given below:

**Analog Output Interface:**

- S-video
- RGB (Not supported)
- YPBPR (Not supported)
- Composite (Not supported)

**Digital Input Interface:**

- 8-bit BT656, Embedded Sync
- 8-bit BT656, External Sync (Not supported)
- RGB 24 bit (Not supported)
- RGB 555 (Not supported)
- RGB 565 (Not supported)
- Color Index (Not supported)

SAA7105 video decoder is an independent interface which is being configured from the VPORT driver. SAA7105 is I2C slave device. SAA7105 driver configures SAA7105 device using I2C interface.

### 5.3.1 Interface Functions

For display devices, EDC driver is having one intermediate layer driver (decoder driver), which takes care about lower layer encoder function calls. This intermediate decoder exports its function table pointer through *edcEncoder_Fxns* global variable as defined below:

```
EDC_Fxns edcEncoder_Fxns = {
    edcEncoder_open,
    edcEncoder_close,
    edcEncoder_ctrl
};
```

To use SAA7105, application shall pass this function table pointer as part of device parameters (*edcTbl[2]* of *VPORT_PortParams*) during driver registration using BIOS TCI file. This will associate the EDC driver instance with the corresponding VPORT driver instance.

As shown in *Figure 3*, when application calls *FVID_create*, VPORT driver will internally call *edcEncoder_open* function. String *"SAA7105"* should be passed as argument to this function to open SAA7105 for SD display. This will power on SAA7105 device and initialize I2C driver for serial communication. Also THS8200 will be put into BT.656 bypass mode.

To configure SAA7105, application has to call *FVID_control* function with *VPORT_CMD_EDC_BASE + SAA7105 IOCTL* (as shown in below table) as command. This will internally call *SAA7105_ctrl* function through *edcEncoder_ctrl* function. Once the VPORT driver deletes the channel, it will delete the SAA7105 and THS8200 driver instances and close the I2C driver as well.

SAA7105 driver provides support for different IOCTL commands as shown below. Application can call *FVID_control* with one of below specified IOCTL command and corresponding argument to configure SAA7105.

| IOCTL Command | Argument | Description |
|---|---|---|
| *EDC_CONFIG* | Pointer to structure variable of type *SAA7105_ConfParams* | Configures SAA7105 encoder for display operation |
| *EDC_RESET* | NULL | Resets SAA7105 encoder |
| *SAA7105_POWERDOWN* | NULL | Power down SAA7105 |
| *SAA7105_POWERUP* | NULL | Power up SAA7105 |
| *SAA7105_ENABLE_COLORBAR* | Boolean to enable or disable color bar<br>TRUE = Enable color bar<br>FALSE = Disable color bar | Enables or disables internal color bar of SAA7105 |

### 5.3.2 Data Structures

❖ **SAA7105_ConfParams**

*"saa7105.h"* file contains *SAA7105_ConfParams* data structure that is passed while calling *EDC_CONFIG* IOCTL for SAA7105 from the application. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *aFmt* | Type of enum *SAA7105_AnalogFormat*, indicates analog output interface |
| *mode* | Type of enum *SAA7105_Mode*, indicates analog output standard |
| *iFmt* | Type of enum *SAA7105_InputFormat*, indicates digital input format |
| *enableSlaveMode* | Boolean to select master or slave mode<br>TRUE = Slave mode<br>FALSE = Master mode (Not supported) |
| *enableBT656Sync* | Boolean to select embedded or external sync for digital input<br>TRUE = Embedded sync<br>FALSE = External sync (Not supported) |

### 5.3.3 Enumerations

❖ **SAA7105_AnalogFormat**

*"saa7105.h"* file contains *SAA7105_AnalogFormat* enum that is passed while calling *EDC_CONFIG* IOCTL for SAA7105 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *SAA7105_AFMT_SVIDEO* | S-video analog output |
| *SAA7105_AFMT_RGB* | Component analog RGB output (Not supported) |
| *SAA7105_AFMT_YPBPR* | Component analog YPBPR output (Not supported) |
| *SAA7105_AFMT_COMPOSITE* | Composite analog output (Not supported) |

❖ **SAA7105_InputFormat**

*"saa7105.h"* file contains *SAA7105_InputFormat* enum that is passed while calling *EDC_CONFIG* IOCTL for SAA7105 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *SAA7105_IFMT_RGB24_YCBCR444* | 24/30 bit RGB888 or YUV444 (Not supported) |
| *SAA7105_IFMT_RGB555* | 15 bit RGB555 (Not supported) |
| *SAA7105_IFMT_RGB565* | 16 bit RGB565 (Not supported) |
| *SAA7105_IFMT_YCBCR422_NONEINTERLACED* | YUV 4:2:2 (Not supported) |
| *SAA7105_IFMT_YCBCR422_INTERLACED* | YUV 4:2:2 |
| *SAA7105_IFMT_COLOR_INDEX* | Color look-up table index mode (Not supported) |

### ❖ SAA7105_Mode

*"saa7105.h"* file contains *SAA7105_Mode* enum that is passed while calling *EDC_CONFIG* IOCTL for SAA7105 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *SAA7105_MODE_NTSC720* | SDTV NTSC 720x480 interlaced at 30fps |
| *SAA7105_MODE_PAL720* | SDTV PAL 720x576 interlaced at 25fps |
| *SAA7105_MODE_VGA* | VESA 640x480 (Not supported) |
| *SAA7105_MODE_SVGA* | VESA 800x600 (Not supported) |
| *SAA7105_MODE_XGA* | VESA 1024x768 (Not supported) |
| *SAA7105_MODE_HD480P60F* | HDTV 720x480 progressive at 60fps (Not supported) |
| *SAA7105_MODE_HD720P60F* | HDTV 1280x720 progressive at 60fps (Not supported) |
| *SAA7105_MODE_HD1080I30F* | HDTV 1920x1080 interlaced at 30fps (Not supported) |
| *SAA7105_MODE_HD720P24F* | HDTV 1280x720 progressive at 24fps (Not supported) |

## 5.4 THS8200 Encoder

THS8200 can accept a variety of digital input formats, in both 4:4:4 and 4:2:2 format, over a 3 ×10-bit, 2 ×10-bit or 1 ×10-bit interface. The device synchronizes to incoming video data either through dedicated Hsync or Vsync inputs or through extraction of the sync information from embedded sync (SAV/EAV) codes inside the video stream. Alternatively, when configured for generating PC graphics output, THS8200 also provides a master timing mode in which it requests video data from an external (memory) source.

In DM648 EVM, THS8200 encoder is used for EDTV, HDTV and VESA resolution displays. SAA7105 encoder which is connected to the VPORT through THS8200 BT.656 bypass port is used for NTSC/PAL SD resolution displays.

THS8200 input and output interface details are given below:

**Analog Output Interface:**

- RGB without sync (used for VESA/PC graphics)
- RGB with sync on G
- RGB with sync on all
- YPbPr with sync on Y (used for HDTV/EDTV)
- YPbPr with sync on all (used for HDTV/EDTV)

**Digital Input Interface:**

- 8/10-bit YCbCr 4:2:2 with Embedded Sync (Not supported)
- 16/20-bit YCbCr 4:2:2 with External Sync (Not supported)
- 16/20-bit YCbCr 4:2:2 with Embedded Sync
- 24/30 bit YCbCr/RGB 4:4:4 (Not supported)
- 16 bit RGB 565 with External Sync (Used for VESA displays)
- 15 bit RGB 555 with External Sync (Not supported)

THS8200 video encoder is an independent interface which is being configured from the VPORT driver. THS8200 is I2C slave device. THS8200 driver configures THS8200 device using I2C interface.

### 5.4.1 Interface Functions

For display devices, EDC driver is having one intermediate layer driver (edcEncoder driver), which takes care about lower layer encoder function calls. This intermediate edcEncoder exports its function table pointer through *edcEncoder_Fxns* global variable as defined below:

```
EDC_Fxns edcEncoder_Fxns = {
    edcEncoder_open,
    edcEncoder_close,
    edcEncoder_ctrl
};
```

To use THS8200, application shall pass this function table pointer as part of device parameters (*edcTbl[2]* of *VPORT_PortParams*) during driver registration using BIOS TCI file. This will associate the EDC driver instance with the corresponding VPORT driver instance.

As shown in *Figure 3*, when application calls *FVID_create*, VPORT driver will internally call *edcEncoder_open* function. String *"THS8200"* should be passed as argument to this function to open THS8200 for HD or VESA display. This will power on THS8200 device and initialize

I2C driver for serial communication. Also THS8200 will be taken out of BT.656 bypass mode and SAA7105 encoder will be power downed.

To configure THS8200, application has to call *FVID_control* function with *VPORT_CMD_EDC_BASE* + THS8200 IOCTL (as shown in below table) as command. This will internally call *THS8200_ctrl* function through *edcEncoder_ctrl* function. Once the VPORT driver deletes the channel, it will delete the THS8200 and SAA7105 driver instances and close the I2C driver as well.

THS8200 driver provides support for different IOCTL commands as shown below. Application can call *FVID_control* with one of below specified IOCTL command and corresponding argument to configure THS8200.

| IOCTL Command | Argument | Description |
|---|---|---|
| *EDC_CONFIG* | Pointer to structure variable of type *THS8200_ConfParams* | Configures THS8200 encoder for display operation |
| *EDC_RESET* | NULL | Resets THS8200 encoder |
| *THS8200_POWERDOWN* | NULL | Power down THS8200 |
| *THS8200_POWERUP* | NULL | Power up THS8200 |
| *THS8200_ENABLE_COLORBAR* | Pointer to structure variable of type *THS8200_ColorbarParams* | Enables or disables internal color bar only in PC Graphics mode (VESA). THS8200 should be configured as master for this. |
| *THS8200_ENABLE_BYPASS* | Boolean<br>TRUE - Enable bypass mode<br>FALSE - Disable bypass mode | To enable or disable BT.656 bypass mode. Note that the application doesn't have to call this, as this is taken care by the *edcEncoder* driver if SAA7105 is configured for SD display. |

### 5.4.2 Data Structures

❖ **THS8200_ConfParams**

*"ths8200.h"* file contains *THS8200_ConfParams* data structure that is passed while calling *EDC_CONFIG* IOCTL for THS8200 from the application. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *mode* | Type of enum *THS8200_Mode*, indicates output video resolution for THS8200 |
| *iFmt* | Type of enum *THS8200_InputFormat*, indicates digital input format |
| *aFmt* | Type of enum *THS8200_AnalogFormat*, indicates analog output format |
| *enableSlaveMode* | Boolean to select master or slave mode |

| | |
|---|---|
| | TRUE = Slave mode<br>FALSE = Master mode (Not supported) |
| *enableBT656Sync* | Boolean to select embedded or external sync for digital input.<br>TRUE = Embedded sync (Used for HDTV resolutions)<br>FALSE = External sync (Used for VESA resolutions) |

❖ **THS8200_ColorbarParams**

*"ths8200.h"* file contains *THS8200_ColorbarParams* data structure that is passed while calling *THS8200_ENABLE_COLORBAR* IOCTL from the application. Note: Since the current driver does not support slave mode of operation, internal color bar for VESA master mode is not supported. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *enableColorbar* | Boolean to enable or disable color bar<br>TRUE = Enable color bar<br>FALSE = Disable color bar |
| *colorbarWidth* | Width for each color bar |

### 5.4.3 Enumerations

❖ **THS8200_AnalogFormat**

*"ths8200.h"* file contains *THS8200_AnalogFormat* enum that is passed while calling *EDC_CONFIG* IOCTL for THS8200 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *THS8200_AFMT_RGB_NO_SYNC* | RGB without sync – used for VESA display |
| *THS8200_AFMT_RGB_SYNC_ON_GREEN* | RGB with sync on Green channel |
| *THS8200_AFMT_RGB_SYNC_ON_ALL* | RGB with sync on all R, G, B channels |
| *THS8200_AFMT_YPBPR_SYNC_ON_Y* | YPbPr with sync on Y channel – used for HDTV display |
| *THS8200_AFMT_YPBPR_SYNC_ON_ALL* | YPbPr with sync on all Y, Pb, Pr channels – used for HDTV display |

❖ **THS8200_InputFormat**

*"ths8200.h"* file contains *THS8200_InputFormat* enum that is passed while calling *EDC_CONFIG* IOCTL for THS8200 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *THS8200_IFMT_YCBCR422_10_BIT* | 10 bit YUV 4:2:2 (Not supported) |
| *THS8200_IFMT_YCBCR422_20_BIT* | 16/20 bit YUV422 |
| *THS8200_IFMT_YCBCR444* | 24/30 bit YUV444 (Not supported) |

| | |
|---|---|
| *THS8200_IFMT_RGB444* | 24/30 bit RGB444 (Not supported) |
| *THS8200_IFMT_RGB565* | 16 bit RGB565 |
| *THS8200_IFMT_RGB555* | 15 bit RGB555 (Not supported) |

### ❖ THS8200_Mode

*"ths8200.h"* file contains *THS8200_Mode* enum that is passed while calling *EDC_CONFIG* IOCTL for THS8200 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *THS8200_MODE _480P_60HZ* | EDTV 720x480 progressive at 60fps (Not supported) |
| *THS8200_MODE _576p_50HZ* | EDTV 720x576 progressive at 50fps (Not supported) |
| *THS8200_MODE _720p_60HZ* | HDTV 1280x720 progressive at 60fps |
| *THS8200_MODE _720p_50HZ* | HDTV 1280x720 progressive at 50fps (Not supported) |
| *THS8200_MODE _1080I_60HZ* | HDTV 1920x1080 interlaced at 30fps |
| *THS8200_MODE _1080I_50HZ* | HDTV 1920x1080 interlaced at 25fps |
| *THS8200_MODE _VGA_60HZ* | VESA 640x480 at 60Hz |
| *THS8200_MODE _VGA_72HZ* | VESA 640x480 at 72Hz |
| *THS8200_MODE _VGA_75HZ* | VESA 640x480 at 75Hz |
| *THS8200_MODE _VGA_85HZ* | VESA 640x480 at 85Hz |
| *THS8200_MODE _SVGA_60HZ* | VESA 800x600 at 60Hz |
| *THS8200_MODE _SVGA_72HZ* | VESA 800x600 at 72Hz |
| *THS8200_MODE _SVGA_75HZ* | VESA 800x600 at 75Hz |
| *THS8200_MODE _SVGA_85HZ* | VESA 800x600 at 85Hz |
| *THS8200_MODE _XGA_60HZ* | VESA 1024x768 at 60Hz |
| *THS8200_MODE _XGA_70HZ* | VESA 1024x768 at 70Hz |
| *THS8200_MODE _XGA_75HZ* | VESA 1024x768 at 75Hz (Not supported) |
| *THS8200_MODE _XGA_85HZ* | VESA 1024x768 at 85Hz (Not supported) |
| *THS8200_MODE _SXGA_60HZ* | VESA 1280x1024 at 60Hz |
| *THS8200_MODE _SXGA_72HZ* | VESA 1280x1024 at 72Hz (Not supported) |
| *THS8200_MODE _SXGA_75HZ* | VESA 1280x1024 at 75Hz (Not supported) |
| *THS8200_MODE _SXGA_85HZ* | VESA 1280x1024 at 85Hz (Not supported) |

## 5.5 TVP7000 Decoder

TVP7000 decoder is a complete solution for digitizing video and graphic signals in RGB or YPbPr color spaces. The device supports pixel rates up to 150 MHz.

DM648 EVM contains one TVP7000 capable of capturing HDTV modes (720p and 1080i). VESA capture is not supported in the current DM648 EVM.

TVP7000 input and output interface details are given below:

**Analog Input Interface:**

- HDTV YPbPr Analog format
- VESA RGB Analog format (Not supported)

**Digital Output Interface:**

- YCbCr 422 - 16-bit YCbCr 4:2:2
- RGB/YCbCr 444 - 24-bit RGB 4:4:4 (Not supported)

TVP7000 video decoder is an independent interface which is being configured from the VPORT driver. TVP7000 is I2C slave device. TVP7000 driver configures TVP7000 device using I2C interface.

### 5.5.1 Interface Functions

TVP7000 exports its function table pointer through *TVP7000_Fxns* global variable as defined below:

```
EDC_Fxns TVP7000_Fxns = {
    TVP7000_open,
    TVP7000_close,
    TVP7000_ctrl
};
```

To use TVP7000, application shall pass this function table pointer as part of device parameters (*edcTbl[2]* of *VPORT_PortParams*) during driver registration using BIOS TCI file. This will associate the EDC driver instance with the corresponding VPORT driver instance.

As shown in *Figure 3*, when application calls *FVID_create*, VPORT driver will internally call *TVP7000_open* function. This will power on TVP7000 device and initialize I2C driver for serial communication. Strings *"0"* should be passed as argument to *TVP7000_open* function to open the TVP7000 decoder.

To configure TVP7000, application has to call *FVID_control* function with *VPORT_CMD_EDC_BASE* + TVP7000 IOCTL (as shown in below table) as command. This will internally call *TVP7000_ctrl* function. Once the VPORT driver deletes the channel, it will delete the TVP7000 driver instance and close the I2C driver as well.

TVP7000 driver provides support for different IOCTL commands as shown below. Application can call *FVID_control* with one of below specified IOCTL command and corresponding argument to configure TVP7000.

⚐ When TVP7000 is powered down, it stops generating clock. When the application is reloaded after powering down TVP7000, it hangs because resetting VPORT requires the clock to be active. To avoid this TVP7000 should be powered up using Gel file in this scenario.

| IOCTL Command | Argument | Description |
|---|---|---|
| *EDC_CONFIG* | Pointer to structure variable of type *TVP7000_ConfParams* | Configures TVP7000 decoder for capture operation |
| *EDC_RESET* | NULL | Resets TVP7000 decoder |
| *TVP7000_POWERDOWN* | NULL | Power down TVP7000 |
| *TVP7000_POWERUP* | NULL | Power up TVP7000 |
| *TVP7000_SET_ANALOG_CHAN* | Pointer to structure variable of type *TVP7000_AnalogChanParams* | Selects analog input channels for TVP7000 decoder |

### 5.5.2 Data Structures

❖ **TVP7000_ConfParams**

*"tvp7000.h"* file contains *TVP7000_ConfParams* data structure that is passed while calling *EDC_CONFIG* IOCTL for TVP7000 from the application. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *mode* | Type of enum *TVP7000_Mode*, indicates input analog video mode |
| *oFmt* | Type of enum *TVP7000_OutputFormat*, indicates output digital video format |

❖ **TVP7000_AnalogChanParams**

*"tvp7000.h"* file contains *TVP7000_AnalogChanParams* data structure that is passed while calling *TVP7000_SET_ANALOG_CHAN* IOCTL from the application. The members of this structure are explained below:

| Structure Members | Description |
|---|---|
| *chanR* | Type of enum *TVP7000_AnalogChan*, analog channel selection for R (Pr) input |
| *chanG* | Type of enum *TVP7000_AnalogChan*, analog channel selection for G (Y) input |
| *chanB* | Type of enum *TVP7000_AnalogChan*, analog channel selection for B (Pb) input |
| *chanSOG* | Type of enum *TVP7000_AnalogChan*, analog channel selection for SOG input |
| *chanHsync* | Analog channel selection for Hsync input (Not supported) *TVP7000_AFMT_CHAN_A* or *TVP7000_AFMT_CHAN_B* |
| *chanVsync* | Analog channel selection for Vsync input (Not supported) *TVP7000_AFMT_CHAN_A* or *TVP7000_AFMT_CHAN_B* |

### 5.5.3 Enumerations

❖ **TVP7000_OutputFormat**

*"tvp7000.h"* file contains *TVP7000_OutputFormat* enum that is passed while calling *EDC_CONFIG* IOCTL for TVP7000 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *TVP7000_OUTMODE_24BIT_RGB_YCBCR_444* | Selection for RGB/YUV 4:4:4 output (Not supported) |
| *TVP7000_OUTMODE_16BIT_YCBCR_422* | YUV 4:2:2 output |

❖ **TVP7000_AnalogChan**

*"tvp7000.h"* file contains *TVP7000_AnalogChan* enum that is passed while calling *TVP7000_SET_ANALOG_CHAN* IOCTL from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *TVP7000_AFMT_CHAN_A* | Decoder channel A selection |
| *TVP7000_AFMT_CHAN_B* | Decoder channel B selection (Not supported) |
| *TVP7000_AFMT_CHAN_C* | Decoder channel C selection (Not supported) |

❖ **TVP7000_Mode**

*"tvp7000.h"* file contains *TVP7000_Mode* enum that is passed while calling *EDC_CONFIG* IOCTL for TVP7000 from the application. The members of this enum are explained below:

| Enum Members | Description |
|---|---|
| *TVP7000_MODE_480I_60HZ* | SDTV 720x480 interlaced at 30fps (Not supported) |
| *TVP7000_MODE_576I_50HZ* | SDTV 720x576 interlaced at 25fps (Not supported) |
| *TVP7000_MODE_480P_60HZ* | EDTV 720x480 progressive at 60fps (Not supported) |
| *TVP7000_MODE_576P_50HZ* | EDTV 720x576 progressive at 50fps (Not supported) |
| *TVP7000_MODE_720p_60HZ* | HDTV 1280x720 progressive at 60fps |
| *TVP7000_MODE_720p_50HZ* | HDTV 1280x720 progressive at 50fps (Not supported) |
| *TVP7000_MODE_1080I_60HZ* | HDTV 1920x1080 interlaced at 30fps |
| *TVP7000_MODE_1080I_50HZ* | HDTV 1920x1080 interlaced at 25fps |
| *TVP7000_MODE_1080p_60HZ* | HDTV 1920x1080 progressive at 60fps (Not supported) |
| *TVP7000_MODE_1080p_50HZ* | HDTV 1920x1080 progressive at 50fps (Not supported) |
| *TVP7000_MODE_VGA_60HZ* | VESA 640x480 at 60Hz (Not supported) |
| *TVP7000_MODE_VGA_72HZ* | VESA 640x480 at 72Hz (Not supported) |
| *TVP7000_MODE_VGA_75HZ* | VESA 640x480 at 75Hz (Not supported) |

| | |
|---|---|
| *TVP7000_MODE_VGA_85HZ* | VESA 640x480 at 85Hz (Not supported) |
| *TVP7000_MODE_SVGA_60HZ* | VESA 800x600 at 60Hz (Not supported) |
| *TVP7000_MODE_SVGA_72HZ* | VESA 800x600 at 72Hz (Not supported) |
| *TVP7000_MODE_SVGA_75HZ* | VESA 800x600 at 75Hz (Not supported) |
| *TVP7000_MODE_SVGA_85HZ* | VESA 800x600 at 85Hz (Not supported) |
| *TVP7000_MODE_XGA_60HZ* | VESA 1024x768 at 60Hz (Not supported) |
| *TVP7000_MODE_XGA_70HZ* | VESA 1024x768 at 70Hz (Not supported) |
| *TVP7000_MODE_XGA_75HZ* | VESA 1024x768 at 75Hz (Not supported) |
| *TVP7000_MODE_XGA_85HZ* | VESA 1024x768 at 85Hz (Not supported) |
| *TVP7000_MODE_SXGA_60HZ* | VESA 1280x1024 at 60Hz (Not supported) |
| *TVP7000_MODE_SXGA_70HZ* | VESA 1280x1024 at 70Hz (Not supported) |
| *TVP7000_MODE_SXGA_75HZ* | VESA 1280x1024 at 75Hz (Not supported) |
| *TVP7000_MODE_SXGA_85HZ* | VESA 1280x1024 at 85Hz (Not supported) |

# 6 Porting Description

This section describes the porting of VPORT driver on different TI platforms.

## 6.1 Porting of DM648 VPORT driver to different DM648 EVM

This section provides description for porting of VPORT driver on different DM648 EVM, having decoders and encoders other than EVMDM648 board. The figure below shows VPORT device driver architecture and changes those are required at the driver layers while porting VPORT device driver to other EVM.

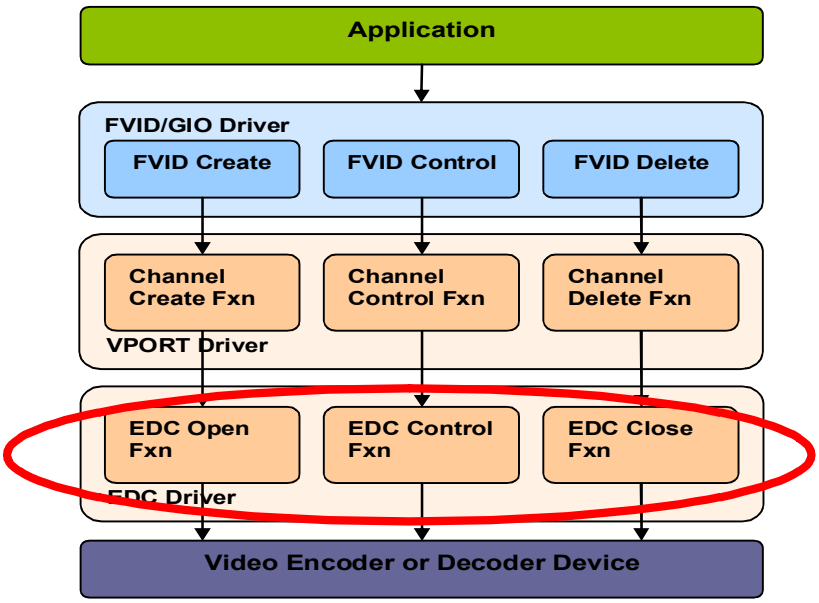


**Figure 4.** Porting of DM648 VPORT driver to different DM648 EVM

User should take care of below mentioned points while porting DM648 VPORT driver on different EVM:

- If any encoders and decoders are different than TVP7000, TVP5154, THS8200 & SAA7105, EDC driver for respective encoder or decoder should be developed. The interface of EDC driver should be same as described in EDC section.

- If encoders and decoders are same as EVMDM648, but if their hardware interface with Video Port is different than EVMDM648 then corresponding modifications should be done in EDC driver. For example, In EVMDM648, SAA7105 is connected with VPORT via THS8200 bypass mode. If this interface changes then corresponding modifications should be done in EDC driver.

## 6.2 Porting of DM648 VPORT driver to different processor having same VPORT IP as DM648

This section provides guidance for porting of VPORT driver on the DSP other than DM648 with same VPORT IP.

Below mentioned points should be taken care of while porting VPORT driver on different processor with EDMA3 IP same as DM648:

- Modify CSLR files for VPORT base addresses and EDMA events.
- Modify PortObj structure in VPORT driver as per number of video ports in new processor.
- Change IRQ id in vport.h file which is used for VPORT interrupts.
- If ECM module is different than DM648 then ECM calls should be modified accordingly in VPORT driver.
- If I2C layer functionality is changed in new processor then user shall modify I2C interface for EDC driver.
- If EDMA IP in new processor is different than DM648 EDMA3 then data transfer logic using EDMA should be modified in VPORT driver.
- Refer to section 6.1 for modifications required in EDC driver.

# 7 Example Applications

This section describes the example applications that are included in the package. These sample application can be run as is for quick demonstration, but the user will benefit most by using these application as sample source code in developing new applications.

## 7.1 Writing Applications for VPORT

This section provides guidance to user for writing their own application for VPORT capture and display drivers

### 7.1.1 File Inclusion

To write sample application user has to include following header files in the application:

1. **fvid.h**

   This file contains FVID layer macros and structures. These macros are wrapper macros specifically for Video above GIO Layer.

2. **vport.h**

   This file contains video port parameters which are passed to driver at the time of VPORT driver registration with BIOS. It also contains structures related to interrupt generation and configuration.

3. **vportcap.h**

   This file contains configuration structures and defines for capture channel configuration.

4. **vportdis.h**

   This file contains configuration structures and defines for display channel configuration.

5. **edc.h**

   This file contains EDC specific defines, data types and function pointer table structure.

6. **tvp5154.h**

   This file contains the interfaces, data types and symbolic definitions that are needed by the application to configure the TVP5154 video decoder. This header files needs to be added at the application only if the input to VPORT module is from TVP5154 video decoder.

7. **tvp7000.h**

   This file contains the interfaces, data types and symbolic definitions that are needed by the application to configure the TVP7000 video decoder. This header files needs to be added at the application only if the input to VPORT module is from TVP7000 video decoder.

**8. edcEncoder.h**

This file contains external function table declaration for encoder device functions. This header files needs to be added at the application if display is required.

**9. saa7105.h**

This file contains the interfaces, data types and symbolic definitions that are needed by the application to configure the SAA7105 video encoder. This header files needs to be added at the application only if the video output is configured from SAA7105 video encoder.

**10.ths8200.h**

This file contains the interfaces, data types and symbolic definitions that are needed by the application to configure the THS8200 video encoder. This header files needs to be added at the application only if the video output is configured from THS8200 video encoder.

### 7.1.2 Buffer Allocation and Management

Frame buffers containing video data are allocated and initially owned by the drivers when application creates channel. The number of frame buffers that drivers allocate is run-time configurable by application with a minimum requirement of triple buffering.

Before allocation, drivers calculate the size of each buffer based on the channel configuration parameters. For example, the size of a buffer that can hold an entire NTSC video frame is 720x480x2.

If horizontal scaling is enabled, however, the size would be halved. Frame buffers are exchanged among the application and the drivers by using the *FVID_alloc (FVID_dequeue)*, *FVID_free (FVID_queue)* and *FVID_exchange* functions. The buffer management strategies, however, are different in the capture and display drivers, as showed in Figure 4 and Figure 5 below.

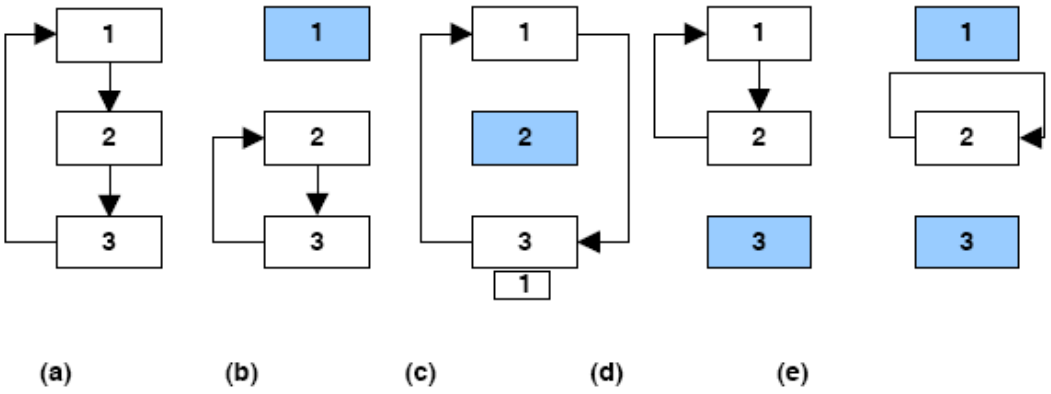


**Figure 5.** Capture Driver Buffer Management

In the case of capture, all buffers are initially in the free queue and the driver cycles through them in a circular fashion. This is illustrated in Figure 4(a).
When the application calls *FVID_alloc* and grabs the buffer with the most recent data from the driver, the driver then cycles through the rest of buffers. This is illustrated in Figure 4 from (a) to (b) and from (b) to (e).

When the application calls *FVID_free (FVID_queue)*, an empty buffer is returned by the application to the driver's free queue. This is illustrated in Figure 4 from (b) to (a) or from (e) to (b). When the application calls *FVID_exchange*, an empty buffer is returned by the application to the driver's free queue, and a buffer with the most recent data is given to the application. This is equivalent to calling *FVID_free (FVID_queue)* and *FVID_alloc* sequentially, as shown in Figure 4 from (b) to (c) and from (c) to (d).

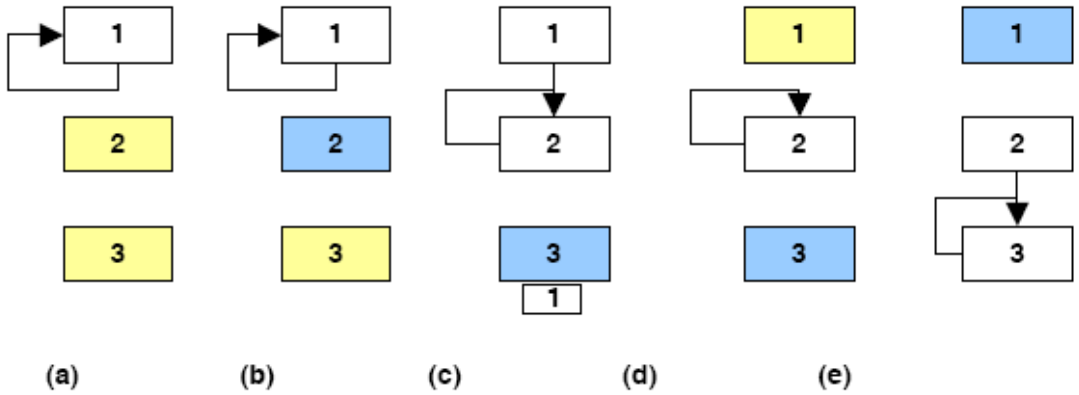


**Figure 6.** Display Driver Buffer Management

In the case of display, initially all buffers except one are in the output queue, ready to be grabbed by the application. The driver repeatedly displays the current buffer. This is shown in Figure 5(a). When the application calls *FVID_alloc (FVID_dequeue)*, it gets a buffer from the driver. Application starts to fill data to it while the driver is still displaying its current buffer. This is shown in Figure 5(b) and (d).

When the application calls *FVID_free (FVID_queue)*, it returns a buffer ready for display back to the driver. The driver, in turn, will set this buffer as its current buffer after it completes displaying the previous one. This is shown in Figure 5(b) to (c) to (d). When the application calls *FVID_exchange*, it returns a buffer ready for display back to the drive and it requires an empty buffer from the driver. This is equivalent to calling *FVID_free (FVID_queue)* and *FVID_alloc (FVID_dequeue)* sequentially, as shown in Figure 5(d) to (e).

### 7.1.3 Cache Coherency

Application is responsible to ensure cache coherency of video buffers, as the driver does nothing in this respect. This is because data is typically moved by EDMA between fast on-chip RAM and slow off-chip SD-RAM for faster CPU access. Furthermore, algorithms can use ping-pong buffer schemes to parallel the EDMA transfer and the CPU execution, thus hiding most or all overhead associated with the data movement. If this is the case, cache flush and clean operations can be avoided by aligning the frame buffers to cache line boundaries. However, if the application does access these buffers directly, the application must flush or clean the cache to ensure cache coherency, the EDMA accesses external memory directly through the EMIF, while the CPU goes through the cache when accessing the data.

**Recommended Cache Operation in Application:**

In a simple loopback scenario, the application doesn't have to do any cache operations to ensure cache coherency if buffers are exchanged between drivers. But when the application access the video buffers through CPU say to run an algorithm or to copy capture buffer to display buffer using CPU, then the below cache operations are recommended for proper operation.

- Capture driver

  Before providing a buffer to capture driver, the entire buffer should be invalidated. Below code snippet illustrate this.

```
/* Invalidate the buffer before giving to capture driver */
BCACHE_inv((Uint8 *)frame->frame.iFrm.y1,  FRAME_SIZE, TRUE);

/* Give the old capture frame buffer back to driver and get the
   recently captured frame buffer */
status = FVID_exchange(chanHandle, &frame);
```

- Display driver

  Before providing a buffer to display driver, the entire buffer should be flushed and invalidated. Below code snippet illustrate this.

```
/* Flush and invalidate the processed buffer so that the EDMA reads
   the processed data */
BCACHE_wbInv((Uint8 *) frame->frame.iFrm.y1, FRAME_SIZE, TRUE);

/* Give the captured frame buffer to display driver and get a
   free frame buffer for next capture */
status = FVID_exchange(chanHandle, &frame);
```

## 7.2 Sample Applications

### 7.2.1 Introduction

The sample application is a representative test program. Initialization of VPORT driver is done by calling initialization function from BIOS.

**1. SD Loop back**

The SD loop back application configures capture & display drivers and starts video loop back in NTSC/PAL resolution. By default the sample application captures **one** channel and displays in **NTSC** resolution.

Configuration options are provided (macros defined at the start of *"psp_bios_vport_sd_loopback.c"* file) to change the number of channel to capture and to change loop back for PAL resolution. If the number of capture channel is changed to more than one, the application will display each channel one by one for a specified number of seconds (default to **4** seconds).

**2. SD Compositor**

The SD Compositor application configures capture and display drivers in NTSC/PAL resolutions. By default the sample application captures **four** NTSC channel and displays the four channel in NTSC resolution. This sample application uses new FVID APIs (DM64LC mode of operation).

Configuration options are provided (macros defined at the start of *"psp_bios_vport_sd_compositor.c"* file) to change the number of channel to capture and to change application for PAL resolution. If the number of capture channel is changed to more than one (Maximum 4), the application will club all the capture channels into one frame (Uses half scaling of capture channel depending upon on number of capture channel) and displays all the channels simultaneously in NTSC D1 resolution.

**3. HD Compositor**

The HD Compositor application configures capture drivers in NTSC resolution & display driver in 1080I 60Hz resolution. By default the sample application captures **one** NTSC channel and displays in **1080I 60Hz** resolution. This sample application uses new FVID APIs (DM64LC mode of operation).

Configuration options are provided (macros defined at the start of *"psp_bios_vport_hd_compositor.c"* file) to change the number of channel to capture and to change application for PAL resolution. If the number of capture channel is changed to more than one, the application will club all the capture channels into one frame (Uses half scaling of capture channel depending upon on number of capture channel) and displays all the channels simultaneously in 1080I resolution.

**4. HD Loop back**

The HD loop back application configures capture & display drivers and starts video loop back in 720P/1080I resolution. By default the sample application captures and displays in **1080I 60Hz** resolution.

Configuration options are provided (macros defined at the start of *"psp_bios_vport_hd_loopback.c"* file) to change loop back for 1080I 50 Hz or 720P 60Hz resolutions.

**5. VESA Color bar**

The VESA color bar application configures display drivers and starts video vertical color bar display in VESA resolutions. By default the sample application displays in **VGA 60Hz** resolution. This sample application uses new FVID APIs (DM64LC mode of operation).

Configuration options are provided (macros defined at the start of *"psp_bios_vport_vesa_colorbar.c"* file) to change display for *VGA 72/75/85Hz*, *SVGA 60/72/75/85Hz*, *XGA 60/70Hz* and *SXGA 60Hz* resolutions.

**6. Raw Capture Loop back**

This sample application illustrates the RAW capture capability of VPORT driver. It contains two tasks. Task 1 captures NTSC/PAL video from player/camera through VPORT 0 channel A and displays the same in VPORT 3 in RAW format. The displayed data is loop backed to VPORT 4 channel A using external cables through daughter card. Task 2 captures the loop backed data in RAW format and displays the video through VPORT 1 which can be viewed in TV.

By default the sample application works in **8-bit** RAW display/capture mode and loop back in **NTSC** resolution.

Configuration options are provided (macros defined at the start of *"psp_bios_vport_raw_capture_loopback.c"* file) to change the RAW display/capture mode to 16-bit and to change loop back for PAL resolution.

❖ **Build procedure**

- Open CCS setup. Import proper CCS configuration file. Set the proper CCS Gel file. Click on "Save & Quit" button and start CCS.
- Open *"<root>\packages\ti\sdo\pspdrivers\system\dm648\bios\evmDM648\video\sample\build\sd_loopback**dm648_evm_vport_st_sd_loopback_sample.pjt***"* for running SD Loop back sample application.

  (OR)

  Open *"<root>\packages\ti\sdo\pspdrivers\system\dm648\bios\evmDM648\video\sample\build\sd_compositor**dm648_evm_vport_st_sd_compositor_sample.pjt***"* for running HD Compositor sample application

  (OR)

  Open *"<root>\packages\ti\sdo\pspdrivers\system\dm648\bios\evmDM648\video\sample\build\hd_compositor**dm648_evm_vport_st_hd_compositor_sample.pjt***"* for running HD Compositor sample application

  (OR)

  Open *"<root>\packages\ti\sdo\pspdrivers\system\dm648\bios\evmDM648\video\sample\build\hd_loopback**dm648_evm_vport_st_hd_loopback_sample.pjt***"* for running HD Loop back sample application

  (OR)

Open "*<root>\packages\ti\sdo\pspdrivers\system\dm648\bios\evmDM648\video\sample\build\vesa_colorbar**dm648_evm_vport_st_vesa_colorbar_sample.pjt***" for running VESA color bar sample application

(OR)

Open "*<root>\packages\ti\sdo\pspdrivers\system\dm648\bios\evmDM648\video\sample\build\raw_capture_loopback**dm648_evm_vport_st_raw_capture_loopback_sample.pjt***" for running RAW capture loop back sample application

- Compile the project using Project->Build
- Following are the dependent libraries/projects to successfully build video application
    - VPORT
    - VPORT EDC
    - I2C
    - PAL_OS
    - SoC specific PAL_SYS
    - EDMA3 DRV
    - EDMA3 RM
    - EDMA3 DRV Sample

❖ **Hardware setup and demo procedure for DM648 SD Loop back**

- Connect XDS emulator to the JTAG connector on EVMDM648 board. Switch on the power supply for board.
- Connect RCA video cables from TVP5154 #1 and TVP5154 #2 decoder channels composite input of EVMDM648 to eight NTSC camera or a DVD Player set in NTSC mode. For default application, only one input channel is sufficient.

  Connect the cables in the following sockets

  - Channel 0 - **J4 bottom RCA jack**
  - Channel 1 - **J4 top RCA jack**
  - Channel 2 – **J3 bottom RCA jack**
  - Channel 3 – **J3 top RCA jack**
  - Channel 4 – **J6 bottom RCA jack**
  - Channel 5 – **J6 top RCA jack**
  - Channel 6 – **J5 bottom RCA jack**
  - Channel 7 – **J5 top RCA jack**
- Connect Svideo cable from SAA7105 output of EVMDM648 (**J13**) to TV.
- Make sure the Video Clock is set to 27 MHz and the EVM mux are set properly for SD operation using the GEL file.
- Load the generated video ".out" file (***dm648_evm_vport_st_sd_loopback_sample.out***) and execute it.
- By default, demo will display video (in Svideo format from **J13**) captured from TVP5154 #1 Channel 0 (**J4 bottom jack**) in NTSC D1 resolution.
- User can change the number of capture channel by changing *"**CFG_NUM_CAP_CHANNEL**"* macro to any number between 1 and 8.
- Below are the other configurable options available in this sample application
  - *"**CFG_VIDEO_MODE**"* – Define this to "*MODE_PAL*" for PAL mode of operation. Default value for this macro is *"MODE_NTSC"*
  - *"**CFG_LOOP_TIME**"* - If *"CFG_NUM_CAP_CHANNEL"* is greater than 1, then after "*CFG_LOOP_TIME"* second loop back operation will switch to the next channel. Default value for this macro is 4 second.
  - *"**CFG_ENABLE_SCALING**"* – Define this to 1 to enable ½ scaling for capture and 2x scaling for display operation. Default value for this macro is 0 (scaling disabled).
  - *"**CFG_NUM_FRAME_COUNT**"* - This sample application will run for *"CFG_NUM_FRAME_COUNT"* amount of frames. After which the application will close. With the current value of 100000 frames, the sample application will run for 55 minute of NTSC video or 66 minute of PAL video. After which the loop back operation will stop.

❖ **Hardware setup and demo procedure for DM648 SD Compositor**

- Connect XDS emulator to the JTAG connector on EVMDM648 board. Switch on the power supply for board.
- Connect RCA video cables from TVP5154 #1 and TVP5154 #2 decoder channels composite input of EVMDM648 to eight NTSC camera or a DVD Player set in NTSC mode. For default application, four input channels are required.

  Connect the cables in the following sockets

  - Channel 0 - **J4 bottom RCA jack**
  - Channel 1 - **J4 top RCA jack**
  - Channel 2 – **J3 bottom RCA jack**
  - Channel 3 – **J3 top RCA jack**
- Connect Svideo cable from SAA7105 output of EVMDM648 (**J13**) to TV.
- Make sure the Video Clock is set to 27 MHz and the EVM mux are set properly for SD operation using the GEL file.
- Load the generated video ".out" file (***dm648_evm_vport_st_sd_compositor_sample.out***) and execute it.
- By default, demo will display video (in Svideo format from **J13**) captured from all the four channels in a single frame in NTSC D1 resolution.
- User can change the number of capture channel by changing *"**CFG_NUM_CAP_CHANNEL**"* macro to any number between 1 and 4.
- Below are the other configurable options available in this sample application
  - *"**CFG_VIDEO_MODE**"* – Define this to "*MODE_PAL*" for PAL mode of operation. When PAL mode is selected, capture and display drivers will be configured for PAL resolution. Default value for this macro is *"MODE_NTSC"*.
  - *"**CFG_NUM_FRAME_COUNT**"* - This sample application will run for *"CFG_NUM_FRAME_COUNT"* amount of frames. After which the application will close. With the current value of 100000 frames, the sample application will run for 55 minute of NTSC video or 66 minute of PAL video. After which the loop back operation will stop.

❖ **Hardware setup and demo procedure for DM648 HD Compositor**

- Connect XDS emulator to the JTAG connector on EVMDM648 board. Switch on the power supply for board.
- Connect RCA video cables from TVP5154 #1 and TVP5154 #2 decoder channels composite input of EVMDM648 to eight NTSC camera or a DVD Player set in NTSC mode. For default application, only one input is sufficient.

  Connect the cables in the following sockets

  - Channel 0 - **J4 bottom RCA jack**
  - Channel 1 - **J4 top RCA jack**
  - Channel 2 – **J3 bottom RCA jack**
  - Channel 3 – **J3 top RCA jack**
  - Channel 4 – **J6 bottom RCA jack**
  - Channel 5 – **J6 top RCA jack**
  - Channel 6 – **J5 bottom RCA jack**
  - Channel 7 – **J5 top RCA jack**
- Connect VGA to Component converter cable from THS8200 output of EVMDM648 (**J14**) to a HDTV.
- Make sure the Video Clock is set to 74.25 MHz and the EVM mux are set properly for HD operation using the GEL file.
- Load the generated video ".out" file (***dm648_evm_vport_st_hd_compositor_sample.out***) and execute it.
- By default, demo will display NTSC D1 resolution video captured from TVP5154 #1 Channel 0 (**J4 bottom jack**) in 1080I 60Hz resolution (through **J15**).
- User can change the number of capture channel by changing ***"CFG_NUM_CAP_CHANNEL"*** macro to any number between 1 and 8. Depending on the number of capture channel, the compositor function will accordingly copy the capture buffers to the display buffer.
- Below are the other configurable options available in this sample application
  - *"**CFG_VIDEO_MODE**"* – Define this to "*MODE_PAL*" for PAL mode of operation. When PAL mode is selected, display driver will be configured for 1080I 50Hz resolution. Default value for this macro is *"MODE_NTSC"*.
  - *"**CFG_NUM_FRAME_COUNT**"* - This sample application will run for *"CFG_NUM_FRAME_COUNT"* amount of frames. After which the application will close. With the current value of 100000 frames, the sample application will run for 55 minute of NTSC video or 66 minute of PAL video. After which the loop back operation will stop.

❖ **Hardware setup and demo procedure for DM648 HD Loop back**

- Connect XDS emulator to the JTAG connector on EVMDM648 board. Switch on the power supply for board.
- Connect Component to VGA converter cable from a HD player to TVP7000 input of EVMDM648 (**J10**).
- Connect VGA to Component converter cable from THS8200 output of EVMDM648 (**J14**) to a HDTV.
- Make sure the Video Clock is set to 74.25 MHz and the EVM mux are set properly for HD capture operation using the GEL file.
- Configuration macro (*CFG_ENABLE_THS7353*) is provided in TVP7000 driver (*tvp7000.c* file) to enable or disable THS7353 HD filter configuration. In production EVM THS7353 HD filter is removed. Hence this option is provided to facilitate the use of TVP7000 driver both in pre-production EVMs and production EVMs. By default this macro is disabled – library is built for production EVM. If the EDC driver has to be used for HD capture operation in pre-production EVMs, then this macro should be set to 1 and the EDC library should be recompiled. Also recompile the sample application to include the generated library.
- Configuration macro (*CFG_ENABLE_HDCAPTURE_WORKAROUND*) is provided in *psp_bios_vport_hd_loopback.c* and *_tvp7000.c* files to enable or disable HD capture software workaround for proper HD capture operation in production EVM. Ensure that this macro is same in both the files. By default both these macros are enabled. If these macros are changed, the VPORT EDC driver and HD Loopback sample application should be recompiled.
- Load the generated video ".out" file (***dm648_evm_vport_st_hd_loopback_sample.out***) and execute it.
- By default, demo will display video captured from TVP7000 in 1080I 60Hz resolution.
- Below are the other configurable options available in this sample application
  - *"**CFG_VIDEO_MODE**"* – Change this to run HD loopback application in 720P 60Hz or 1080I 50Hz resolutions
  - *"**CFG_ENABLE_SCALING**"* – Define this to 1 to enable ½ scaling for capture and 2x scaling for display operation. Default value for this macro is 0 (scaling disabled).
  - *"**CFG_NUM_FRAME_COUNT**"* - This sample application will run for *"CFG_NUM_FRAME_COUNT"* amount of frames. After which the application will close.
  - ***"CFG_ENABLE_HDCAPTURE_WORKAROUND"*** – To enable or disable HD capture software workaround for proper HD capture operation in production EVM.

❖ **Hardware setup and demo procedure for DM648 VESA Color bar**

- Connect XDS emulator to the JTAG connector on EVMDM648 board. Switch on the power supply for board.
- Connect VGA to VGA cable from THS8200 output of EVMDM648 (**J14**) to a PC monitor.
- Make sure the Video Clock is set to required value (see below) and the EVM mux are set properly for VESA display operation using the GEL file. Following are the clock required for various VESA modes
    - VGA 60Hz - 25.175 MHz
    - VGA 72Hz/75Hz - 31.5 MHz
    - VGA 85Hz - 36 MHz
    - SVGA 60Hz - 40 MHz
    - SVGA 72Hz - 50 MHz
    - SVGA 75Hz - 49.5 MHz
    - SVGA 85Hz - 56.25 MHz
    - XGA 60Hz - 65 MHz
    - XGA 70Hz - 75 MHz
    - SXGA 60Hz - 108 MHz
- Load the generated video ".out" file (***dm648_evm_vport_st_vesa_colorbar_sample.out***) and execute it.
- By default, demo will display video color bar in VGA 60Hz resolution.
- Below are the other configurable options available in this sample application
    - *"**CFG_VIDEO_MODE**"* – Change this to run VESA color bar application in *VGA 72/75/85Hz*, *SVGA 60/72/75/85Hz*, *XGA 60/70Hz* and *SXGA 60Hz* resolutions
    - *"**CFG_NUM_FRAME_COUNT**"* - This sample application will run for *"CFG_NUM_FRAME_COUNT"* amount of frames. After which the application will close.

❖ **Hardware setup and demo procedure for DM648 RAW Capture Loop back**

- Connect the daughter card to DM648 EVM (J15, J16 and J25). Connect VPORT 3 and VPORT 4 pins according to the below table using external cables/wires. Connection of VPORT data pins D12 to D19 is not required for 8-bit loopback.

| DC Pin (VPORT 3 Pin) | DC Pin (VPORT 4 Pin) |
|---|---|
| P7:1 (VP3 GND) | Connect to external clock source Ground |
| P7:2 (VP3 CLK0) | Connect to external clock – 27 MHz for 8-bit, 13.5 MHz for 16-bit operation |
| P7:3 (VP3 CTL0) | P14:5 (VP4 CTL0) |
| P7:4 (VP3 D2) | P14:9 (VP4 D2) |
| P7:5 (GND) | P14:7 (GND) |
| P7:6 (GND) | P14:8 (GND) |
| P7:7 (VP3 D3) | P14:10 (VP4 D3) |
| P7:8 (VP3 D4) | P14:11 (VP4 D4) |
| P7:9 (VP3 D5) | P14:12 (VP4 D5) |
| P7:10 (VP3 D6) | P15:1 (VP4 D6) |
| P7:11 (GND) | P15:5 (GND) |
| P7:12 (GND) | P15:6 (GND) |
| P7:13 (VP3 D7) | P15:2 (VP4 D7) |
| P7:14 (VP3 D8) | P15:3 (VP4 D8) |
| P8:1 (VP3 D9) | P15:4 (VP4 D9) |
| P8:2 (VP3 CTL1) | P15:7 (VP4 CTL1) |
| P8:3 (GND) | P15:11 (GND) |
| P8:4 (GND) | P15:12 (GND) |
| P8:5 (VP3 CLK1) | P14:6 (VP3 CLK0) |
| P8:6 (VP3 CTL2) | P15:9 (VP3 CTL2) |
| P8:7 (VP3 D12) | P15:10 (VP4 D12) |
| P8:8 (VP3 D13) | P15:13 (VP4 D13) |
| P8:9 (GND) | P20:3 (GND) |
| P8:10 (GND) | P20:4 (GND) |
| P8:11 (VP3 D14) | P15:14 (VP4 D14) |
| P8:12 (VP3 D15) | P20:1 (VP4 D15) |
| P8:13 (VP3 D16) | P20:2 (VP4 D16) |

| | |
|---|---|
| P8:14 (VP3 D17) | P20:5 (VP4 D17) |
| P14:1 (GND) | P20:9 (GND) |
| P14:2 (GND) | P20:10 (GND) |
| P14:3 (VP3 D18) | P20:6 (VP4 D18) |
| P14:4 (VP3 D19) | P20:7 (VP4 D19) |

- Connect XDS emulator to the JTAG connector on EVMDM648 board. Switch on the power supply for board.
- Connect RCA video cable from TVP5154 #1 decoder channel composite input of EVMDM648 to a NTSC camera or a DVD Player set in NTSC mode.

  Connect the cable in J4 bottom RCA jack
- Connect Svideo cable from SAA7105 output of EVMDM648 (**J13**) to TV.
- Make sure the Video Clock (VPORT 1 display) is set to 27 MHz and the EVM mux are set properly for SD operation using the GEL file.
- Connect VPORT 3 and VPORT 4 to daughter card by running "*Set_Muxes_5VideoPorts_VP34_Dcc*" from GEL menu.
- Make sure the external clock to VPORT 3 display is set to 27 MHz for 8-bit operation or to 13.5 MHz for 16-bit operation.
- Load the generated video ".out" file (***dm648_evm_vport_st_raw_capture_loopback_sample.out***) and execute it.
- By default, demo will display video (in Svideo format from **J13**) captured from TVP5154 #1 Channel 0 (**J4 bottom jack**) in NTSC D1 resolution.
- User can change the RAW capture mode to 16-bit by changing *"**CFG_RAW_LOOPBACK_MODE**"* to *MODE_RAW_16BIT*.
- Below are the other configurable options available in this sample application
  - *"**CFG_VIDEO_MODE**"* – Define this to "*MODE_PAL*" for PAL mode of operation. Default value for this macro is *"MODE_NTSC"*
  - *"**CFG_NUM_FRAME_COUNT**"* - This sample application will run for *"CFG_NUM_FRAME_COUNT"* amount of frames. After which the application will close. With the current value of 100000 frames, the sample application will run for 55 minute of NTSC video or 66 minute of PAL video. After which the loop back operation will stop.

### 7.2.2 Configuration Parameters

This section describes how TVP5154 video decoder chip, SAA7105 video encoder chip and VPORT driver parameters are configured for SD (NTSC) loop back application.

❖ **Video Capture Port Configuration Parameters**

Capture port parameter used during VPORT driver registration with BIOS using TCI files. Capture port is configured for dual channel operation.

```
#define CAP_PARAMS_PORT_DEFAULT {                                    \
   TRUE,                          /* enableDualChan             */  \
   VPORT_POLARITY_ACTIVE_HIGH,    /* vport control pin 1 polarity */  \
   VPORT_POLARITY_ACTIVE_HIGH,    /* vport control pin 2 polarity */  \
   VPORT_POLARITY_ACTIVE_HIGH,    /* vport control pin 3 polarity */  \
   &TVP5154_Fxns,                 /* VPORT Channel A EDC driver */  \
   &TVP5154_Fxns,                 /* VPORT Channel B EDC driver */  \
}
```

❖ **Video Display Port Configuration Parameters**

Display port parameter used during VPORT driver registration with BIOS using TCI files.

```
#define DIS_PARAMS_PORT_DEFAULT {                                    \
   FALSE,                         /* enableDualChan             */  \
   VPORT_POLARITY_ACTIVE_HIGH,    /* vport control pin 1 polarity */  \
   VPORT_POLARITY_ACTIVE_HIGH,    /* vport control pin 2 polarity */  \
   VPORT_POLARITY_ACTIVE_HIGH,    /* vport control pin 3 polarity */  \
   & edcEncoder_Fxns,             /* VPORT Channel A EDC driver */  \
   INV,                           /* VPORT Channel B EDC driver */  \
}
```

❖ **Driver naming convention used for Channel creation**

Application calls *FVID_create* to create and initialize a video driver channel.

The name argument is the name specified for the device when it was created in the configuration file or at run-time. The name contains three fields for Capture video port within it like "/VP0CAPTURE/A/0".

1. "VP0CAPTURE" - name of the video port same as UDEV name
2. "A" - channel of selected VPORT. It can be "A" or "B".
3. "0" – Decoder channel number.

In EVMDM648, for TVP5154 using VP0, this can be 0 (For Channel A) or 1 (Channel B).
In EVMDM648, for TVP5154 using VP2, this can be 2 or 3.
In EVMDM648, for TVP5154 using VP3, this can be 4 or 5.
In EVMDM648, for TVP5154 using VP4, this can be 6 or 7.
In EVMDM648, for TVP7000 using VP0, this value should be 0.

For Display, name argument will be "/VP1DISPLAY/SAA7105" for display using SAA7105 and it will be "/VP1DISPLAY/THS8200" for display using THS8200.

### ❖ VPORT Capture Channel Configuration Parameters

Capture driver is configured to capture data in 8-bit BT656 mode. Here, capture driver is configured in NTSC. For more details, please refer to *psp_bios_vport_capParams.h*

```
#define NTSC720_LINE_SZ                 720
#define NTSC720_NUM_LINES_PER_FLD       (480>>1)

#define CAP_PARAMS_CHAN_EMBEDDED_DEFAULT {                      \
   VPORT_MODE_BT656_8BIT,               /* cmode:3  */          \
   VPORT_FLDOP_FRAME,                   /* fldOp:3  */          \
   VPORT_SCALING_DISABLE,               /* scale:1  */          \
   VPORT_RESMPL_DISABLE,                /* resmpl:1 */          \
   VPORTCAP_BPK_10BIT_ZERO_EXTENDED,    /*bpk10Bit:2 */        \
   VPORTCAP_HRST_SAV,                   /* hCtRst:1  */         \
   VPORTCAP_VRST_EAV_V0,                /* vCtRst:1  */         \
   VPORTCAP_FLDD_DISABLE,               /* fldDect:1 */         \
   VPORTCAP_EXC_DISABLE,                /* fldInv:1 */          \
   VPORTCAP_FINV_DISABLE,               /* fldInv:1 */          \
   0,                                   /* fldXStrt1 */         \
   3,                                   /* fldYStrt1 */         \
   0,                                   /* fldXStrt2 */         \
   3,                                   /* fldYStrt2 */         \
   NTSC_LINE_SZ-1,                      /* fldXStop1 */         \
   NTSC_NUM_LINES_PER_FLD+2,            /* fldYStop1 */         \
   NTSC_LINE_SZ-1,                      /* fldXStop2 */         \
   NTSC_NUM_LINES_PER_FLD+2,            /* fldYStop2 */         \
   (NTSC_LINE_SZ>>3),                   /* thrld     */         \
   3,                                   /* numFrmBufs */        \
   128,                                 /* alignment  */        \
   VPORT_FLDS_MERGED,                   /* mergeFlds  */        \
   NULL,                                /* segId      */        \
   TRUE,                                /* autoSyncEnable */    \
   NULL                                 /* EDMA handle */       \
}
```

### ❖ VPORT Display Channel Configuration Parameters

Display driver is configured to display data in 8-bit display mode. Here, display driver is configured in NTSC mode. For more details, please refer to *psp_bios_vport_disParams.h*

```
#define DISPLAY_NTSC_LINE_SZ                720
#define DISPLAY_NTSC_NUM_LINES_PER_FLD      240

#define _DIS_PARAMS_CHAN_NTSC_DEFAULT {                                  \
   VPORT_MODE_BT656_8BIT,                   /* dmode:3      */          \
   VPORT_FLDOP_FRAME,                       /* fldOp:3      */          \
   VPORT_SCALING_DISABLE,                   /* scale:1      */          \
   VPORT_RESMPL_DISABLE,                    /* resmpl:1     */          \
   VPORTDIS_DEFVAL_ENABLE,                  /* defValEn:1   */          \
   VPORTDIS_BPK_10BIT_NORMAL,               /*bpk10Bit:1 */            \
   VPORTDIS_VCTL1_HSYNC,                    /* vctl1Config:2 */         \
   VPORTDIS_VCTL2_VSYNC,                    /* vctl2Config:2 */         \
   VPORTDIS_VCTL3_FLD,                      /* vctl3Config:1 */         \
   VPORTDIS_EXC_DISABLE,                    /* extCtl:3     */          \
   858,                                     /* frmHSize */              \
   525,                                     /* frmVSize */              \
   0,                                       /* imgHOffsetFld1 */        \
   0,                                       /* imgVOffsetFld1 */        \
   DISPLAY_NTSC_LINE_SZ,                    /* imgHSizeFld1   */        \
   DISPLAY_NTSC_NUM_LINES_PER_FLD,          /* imgVSizeFld1   */        \
   0,                                       /* imgHOffsetFld2 */        \
   0,                                       /* imgVOffsetFld2 */        \
   DISPLAY_NTSC_LINE_SZ,                    /* imgHSizeFld2   */        \
   DISPLAY_NTSC_NUM_LINES_PER_FLD,          /* imgVSizeFld2   */        \
   720,                                     /* hBlnkStart     */        \
   856,                                     /* hBlnkStop      */        \
   720,                                     /* vBlnkXStartFld1 */       \
   1,                                       /* vBlnkYStartFld1 */       \
   720,                                     /* vBlnkXStopFld1  */       \
   21,                                      /* vBlnkYStopFld1  */       \
   360,                                     /* vBlnkXStartFld2 */       \
   263,                                     /* vBlnkYStartFld2 */       \
   360,                                     /* vBlnkXStopFld2  */       \
   284,                                     /* vBlnkYStopFld2  */       \
   720,                                     /* xStartFld1 */            \
   1,                                       /* yStartFld1 */            \
   360,                                     /* xStartFld2 */            \
   263,                                     /* yStartFld2 */            \
   736,                                     /* hSyncStart */            \
   800,                                     /* hSyncStop  */            \
   736,                                     /* vSyncXStartFld1  */      \
   4,                                       /* vSyncYStartFld1  */      \
   736,                                     /* vSyncXStopFld1   */      \
   7,                                       /* vSyncYStopFld1   */      \
   307,                                     /* vSyncXStartFld2  */      \
   266,                                     /* vSyncYStartFld2  */      \
   307,                                     /* vSyncXStopFld2   */      \
   269,                                     /* vSyncYStopFld2   */      \
```

```
  16,                                   /* yClipLow        */         \
  235,                                  /* yClipHigh       */         \
  16,                                   /* cClipLow        */         \
  240,                                  /* cClipHigh       */         \
  0x10,                                 /* Default Y value  */        \
  0x80,                                 /* Default Cb value */        \
  0x80,                                 /* Default Cr value */        \
                                                                      \
  VPORTDIS_RGBX_DISABLE,                /* RGB extract disable */     \
  0,                                    /* incPix, for raw mode only */ \
  (DISPLAY_NTSC_LINE_SZ>>3),            /*thrld     */                \
  3,                                    /* numFrmBufs*/               \
  128,                                  /* alignment */               \
  VPORT_FLDS_MERGED,                    /* mergeFlds */               \
  NULL,                                 /* segId     */               \
  NULL                                  /* EDMA handle */             \
}
```

Display driver configuration for 1080I 60Hz.

```
#define DIS_1080I_LINE_SZ           (1920)
#define DIS_1080I_NUM_LINES         (1080)
#define DIS_1080I_NUM_LINES_PER_FLD    (DIS_1080I_NUM_LINES >> 1)

#define DIS_PARAMS_CHAN_1080I_60HZ_DEFAULT {                          \
  VPORT_MODE_YCBCR_8BIT,            /* dmode:3        */              \
  VPORT_FLDOP_FRAME,                /* fldOp:3        */              \
                                                                      \
  VPORT_SCALING_DISABLE,            /* scale:1        */              \
  VPORT_RESMPL_DISABLE,             /* resmpl:1        */             \
  VPORTDIS_DEFVAL_ENABLE,           /* defValEn:1      */             \
  VPORTDIS_BPK_10BIT_NORMAL,        /* bpk10Bit:1      */             \
                                                                      \
  VPORTDIS_VCTL1_HSYNC,             /* vctl1Config:2    */            \
  VPORTDIS_VCTL2_VSYNC,             /* vctl2Config:2    */            \
  VPORTDIS_VCTL3_FLD,               /* vctl3Config:1    */            \
  VPORTDIS_EXC_DISABLE,             /* extCtl:3        */             \
                                                                      \
  2200,                             /* frmHSize        */             \
  1125,                             /* frmVSize        */             \
                                                                      \
  0,                                /* imgHOffsetFld1   */            \
  0,                                /* imgVOffsetFld1   */            \
  DIS_1080I_LINE_SZ,                /* imgHSizeFld1     */            \
  DIS_1080I_NUM_LINES_PER_FLD,      /* imgVSizeFld1     */            \
                                                                      \
  0,                                /* imgHOffsetFld2   */            \
  0,                                /* imgVOffsetFld2   */            \
  DIS_1080I_LINE_SZ,                /* imgHSizeFld2     */            \
  DIS_1080I_NUM_LINES_PER_FLD,      /* imgVSizeFld2     */            \
                                                                      \
  1920,                             /* hBlnkStart      */             \
  2196,                             /* hBlnkStop       */             \
```

```
                                                                    \
  0,                            /* vBlnkXStartFld1  */              \
  1,                            /* vBlnkYStartFld1  */              \
  0,                            /* vBlnkXStopFld1   */              \
  23,                           /* vBlnkYStopFld1   */              \
                                                                    \
  0,                            /* vBlnkXStartFld2  */              \
  563,                          /* vBlnkYStartFld2  */              \
  0,                            /* vBlnkXStopFld2   */              \
  586,                          /* vBlnkYStopFld2   */              \
                                                                    \
  0,                            /* xStartFld1       */              \
  1,                            /* yStartFld1       */              \
                                                                    \
  0,                            /* xStartFld2       */              \
  563,                          /* yStartFld2       */              \
                                                                    \
  2008,                         /* hSyncStart       */              \
  2052,                         /* hSyncStop        */              \
                                                                    \
  2008,                         /* vSyncXStartFld1  */              \
  2,                            /* vSyncYStartFld1  */              \
  2008,                         /* vSyncXStopFld1   */              \
  7,                            /* vSyncYStopFld1   */              \
                                                                    \
  908,                          /* vSyncXStartFld2  */              \
  505,                          /* vSyncYStartFld2  */              \
  908,                          /* vSyncXStopFld2   */              \
  505,                          /* vSyncYStopFld2   */              \
                                                                    \
  16,                           /* yClipLow         */              \
  235,                           /* yClipHigh       */              \
  16,                           /* cClipLow         */              \
  240,                          /* cClipHigh        */              \
                                                                    \
  0x10,                         /* Default Y value  */              \
  0x80,                         /* Default Cb value */              \
  0x80,                         /* Default Cr value */              \
                                                                    \
  VPORTDIS_RGBX_DISABLE,        /* RGB extract disable */           \
  0,                            /* incPix, for raw mode only */     \
                                                                    \
  (DIS_1080I_LINE_SZ >> 4),     /* thrld            */              \
                                                                    \
  3,                            /* numFrmBufs       */              \
  128,                          /* alignment        */              \
  VPORT_FLDS_MERGED,            /* mergeFlds        */              \
                                                                    \
  NULL,                         /* segId            */              \
  NULL                          /* hEdma            */              \
}
```

### ❖ TVP5154 Configuration Parameters

All 8 channels of two TVP5154 video decoders will be configured in NTSC 8-bit BT656, Composite channel A input mode.

```
#define CAP_PARAMS_TVP5154_EMBEDDED_DEFAULT {                    \
   TVP5154_MODE_NTSC,                  /* Mode           */      \
   TVP5154_AFMT_COMPOSITE_A,           /* Analog format  */      \
   TRUE                                /* enableBT656Sync */    \
}
```

### ❖ SAA7105 Configuration Parameters

SAA7105 video encoder will be configured in NTSC 8-bit YUV, S-video output mode.

```
#define DIS_PARAMS_SAA7105_SDTV_EMBEDDED_DEFAULT {               \
   SAA7105_AFMT_SVIDEO,                /* AnalogFormat   */      \
   SAA7105_MODE_NTSC,                  /* Mode           */      \
   SAA7105_IFMT_YCBCR422_INTERLACED,   /* InputFormat    */      \
   TRUE,                               /* enableSlaveMode */    \
   TRUE                                /* enableBT656Sync */    \
}
```

### ❖ THS8200 Configuration Parameters

THS8200 video encoder configuration for 1080I 60Hz display.

```
#define DIS_PARAMS_THS8200_1080I_60HZ_DEFAULT {                  \
   THS8200_MODE_1080I_60HZ,            /* Mode           */      \
   THS8200_IFMT_YCBCR422_20_BIT,       /* InputFormat    */      \
   THS8200_AFMT_YPBPR_SYNC_ON_ALL,     /* AnalogFormat   */      \
   TRUE,                               /* enableSlaveMode */    \
   TRUE                                /* enableBT656Sync */    \
}
```

### ❖ TVP7000 Configuration Parameters

TVP7000 video decoder configuration for 1080I 60Hz capture.

```
#define CAP_PARAMS_TVP7000_1080I_60HZ_DEFAULT {                  \
   TVP7000_MODE_1080I_60HZ,            /* Analog Input Mode     */ \
   TVP7000_OFMT_16BIT_YCBCR_422        /* Digital Output Format */ \
}
```