Application Report
SPRA918A - August 2003

The TMS320DM642 Video Port Mini-Driver

DSP Catalog and Emerging End Equipment

ABSTRACT

This application report describes the usage and design of the video capture and display mini-drivers that work on the TMS320DM642 Evaluation board (EVM). These device drivers are compliant with the DSP/BIOS™ IOM device driver model. The DSP's EDMA is used to transfer data between memory and the TMS320DM642 Video Port. To maximize code reuse and streamline the integration process, both drivers are designed of two distinctive parts: the generic part and the board specific part. The external device control interface (EDC) is defined to bind these two parts together in a plug-and-play manner.

Features:

- Multi-instance (can handle multiple video ports simultaneously).
- Capture driver supports the following modes:
  - Single-channel 8/10-bit BT.656 mode with embedded or external sync
  - Dual-channel 8/10 bit BT.656 mode with embedded sync
  - Single-channel 16/20 bit Y/C mode with embedded or external sync
- Display driver supports the following modes:
  - 8/10-bit BT.656 mode with embedded or external sync.
  - 16/20-bit Y/C mode with embedded or external sync for output formats such as high-definition 480p, 720p and 1080i
  - 8/10/16/20 raw mode with ¾ unpacking, for output formats such as 8/16/24-bit RGB
- Supports enable/disable of video port global interrupt on all defined video port events
- Drivers allocate video frame buffers at initialization time based on configuration parameters passed in by the application
- External Control Interface for seamless integration with different video encoder or decoder devices

Contents

List of Figures

List of Tables

1 Overview

The device driver described here is actually part of an IOM mini-driver. That is, it is implemented as the lower layer of a two-layer device driver model. The upper layer is the FVID module, which is a simple wrapper on top of the DSP/BIOS™ GIO class driver. While GIO provides an independent and generic set of APIs and services for a wide variety of mini-drivers, FVID provides customized APIs for frame video capture and display. Please refer to Appendix B for a detailed description of the FVID APIs.

Figure 1 shows the overall DSP/BIOS device driver architecture. For more information about the IOM device driver model as well as the GIO, SIO/DIO, and PIP/PIO modules, see the References section.

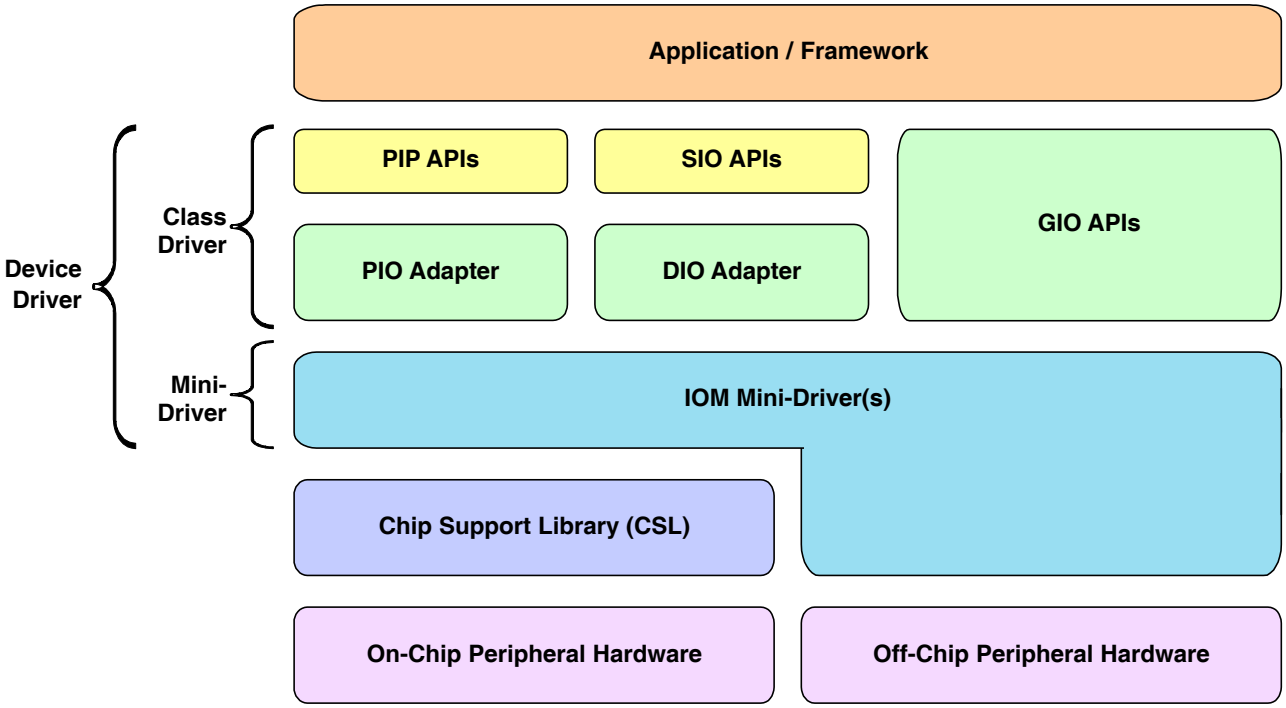

Figure 1. DSP/BIOS IOM Device Driver Model

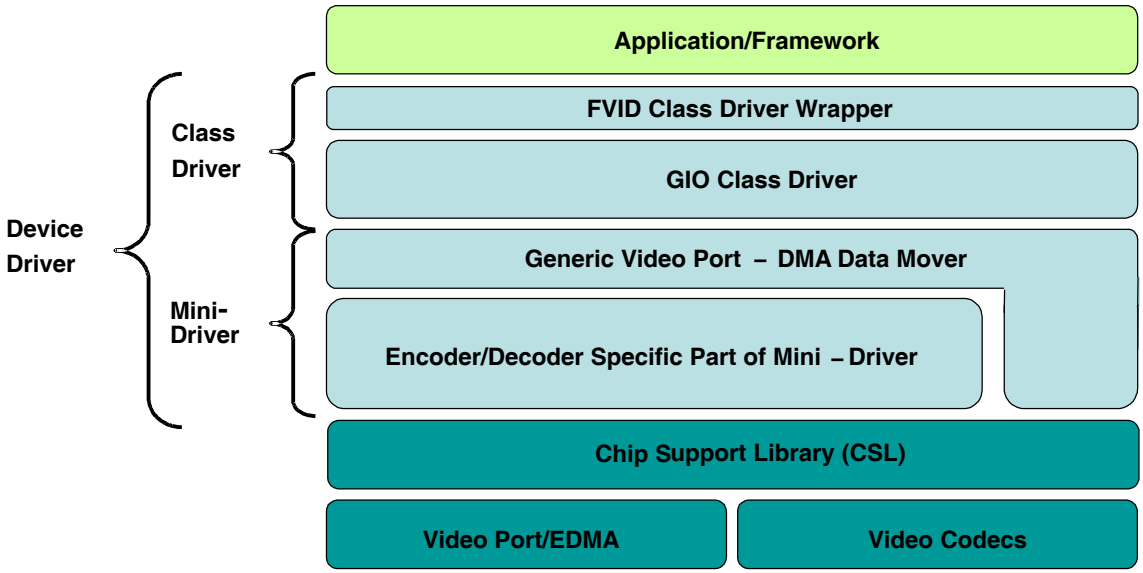

Figure 2. DM642 Video Capture and Display Driver Architecture

Figure 2 shows the architecture of the video capture and display mini-drivers of the DM642 video port. In order to maximize code reusability the DM642 video capture and display mini-drivers are split into two parts: a generic part and a board specific part. In other mini-drivers, such as in the C6x1x EDMA McBSP mini-driver, the class driver interfaces with the board-specific portion of the mini-driver. In the DM642 video mini-drivers, on the other hand, the FVID/GIO class driver interfaces with the generic part of the mini-driver, and the board-specific portion is plugged into the generic portion of mini-driver through an interface called External Device Control or EDC. By mandating that the board-specific portion of the mini-driver be a compliant EDC module, maximum code reuse is achieved by changing only the EDC module of the driver when porting to different platforms.

This application note describes both the generic part and the board-specific part of the DM642 EVM video capture and video display mini-drivers. The generic part of the drivers uses EDMAs to transfer data to and from the video ports. The board-specific part mainly consists of code that in the case of the capture driver, initializes and configures the SAA7115 video decoder, and in the case of the display driver, initializes and configures the SAA7105 video encoder. These EDC-compliant modules set up the video codecs to work together with the video ports to capture or display the desired video data in a specific format. For example, the SAA7105 can be configured to output video data in composite NTSC format or component High-Definition 1080i format or a wide range of other video formats, depending on application requirements. In the mean time, the associated video port must also be configured accordingly.

The board-specific part also requires the EVM and the DM642 DSP to be initialized by calling the EVM642_init() function from the DM642 EVM Board Support Library (BSL), which comes with the EVM. This will set up the EMIF, pin-mux configurations and the I2C controller. An application must link all three libraries necessary in order to function correctly: one from the board-specific part, such as the SAA7115 or the SAA7105, one from the generic VPORTCAP or VPORTDIS part, and one from the BSL. These three libraries are called evm642_saa7115.l64, evm642_vportcap.l64, and evmdm642.l64, respectively for capture, and are called evm642_saa7105.l64, evm642_vportdis.l64, and evmdm642.l64, respectively for display.

2 Usage

2.1 Configuration

To use the capture or display device driver, a device entry must be added and configured in the DSP/BIOS configuration tool. Refer to the *DSP/BIOS Device Driver Developer's Guide* (literature number SPRU616) for more information on how to use the DSP/BIOS configuration tool to configure device drivers. The following are the device configuration settings required to use the capture driver:

- **Init function:** N/A, not used by this driver
- **Function table ptr:** _VPORTCAP_Fxns
- **Function table type:** IOM_Fxns
- **Device id:** 0 or 1 for DM642 EVM:, specify which video port to use
- **Device params ptr:** 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. Setting this pointer to NULL requires that an additional FVID_control call made from the application to initialize the video port. The parameter structure is described below. An example of this structure is the _EVM642_vCapParamsNTSCPort that is defined in the evm642_vcapParamsNTSC.c file for NTSC format video capture.
- **Device global data ptr:** N/A, not used by this driver

The following are the device configuration settings required to use the display driver:

- **Init function:** N/A, not used by this driver
- **Function table ptr:** _VPORTDIS_Fxns
- **Function table type:** IOM_Fxns
- **Device id:** 2 for DM642 EVM: specify which video port is in use
- **Device params ptr:** Same as for the capture driver
- **Device global data ptr:** N/A, not used by this driver

2.2 Device Parameters for Generic Part of the Driver

Please refer to *TMS320C64x DSP Video Port/ VCXO Interpolated Control (VIC) Port Reference Guide* (literature number SPRU629) for a better understanding of the parameters described below.

2.2.1 Port Parameters

```
typedef struct VPORT_PortParams{
 Int  versionId;
 Bool dualChanEnab;
 Uns  vc1Polarity;
 Uns  vc2Polarity;
 Uns  vc3Polarity;
 EDC_Fxns* edcTbl[2];
}VPORT_PortParams;
```

- versionId: Version number of the driver.

- dualChanEnable: Dual channel mode enable (capture only).
- vc1Polarity: polarity of the vctrl1 pin, either active high or active low.
- vc2Polarity: polarity of the vctrl2 pin.
- vc3Polarity: polarity of the vctrl3 pin.
- edcTbl[2]: array of up to two pointers of EDC function tables, one for each channel.

2.2.2 Capture Channel Parameters

```
typedef struct {
 Int   cmode;
 Int   fldOp;
 Int   scale;
 Int   resmpl;
 Int   bpk10Bit;
 Int   hCtRst;
 Int   vCtRst;
 Int   fldDect;
 Int   extCtl;
 Int   fldInv;

 Uint16   fldXStrt1;
 Uint16   fldYStrt1;
 Uint16   fldXStrt2;
 Uint16   fldYStrt2;
 Uint16   fldXStop1;
 Uint16   fldYStop1;
 Uint16   fldXStop2;
 Uint16   fldYStop2;
 Uint16   thrld;

 Int      numFrmBufs;
 Int      alignment;
 Int      mergeFlds;
 Int      segId;
 Int      edmaPri;
 Int      irqId;
}VPORTCAP_Params;
```

The definitions of the bit-fields in the above parameter are mapped to the video port capture control register. The unnamed fields are there to represent the reserved bits in that register.

- **cmode:** capture mode, The following are possible settings defined in vport.h:
  - VPORT_MODE_BT656_8BIT
  - VPORT_MODE_BT656_10BIT
  - VPORT_MODE_RAW_8BIT
  - VPORT_MODE_RAW_10BIT
  - VPORT_MODE_YC_8BIT
  - VPORT_MODE_YC_10BIT
  - VPORT_MODE_RAW_16BIT
  - VPORT_MODE_RAW_20BIT
- fldOp: field and frame operation mode. The following are possible settings defined in vport.h:
  - VPORT_FLDOP_FLD1
  - VPORT_FLDOP_FLD2
  - VPORT_FLDOP_FRAME
  - VPORT_FLDOP_PROGRESSIVE
- **scale:** horizontal ½ scaling enable. The following are possible settings defined in vport.h:
  - VPORT_SCALING_DISABLE
  - VPORT_SCALING_ENABLE
- **resmpl:** chroma horizontal 4:2:2 to 4:2:0 re-sampling enable. The following are possible settings defined in vport.h:
  - VPORT_RESMPL_DISABLE
  - VPORT_RESMPL_DISABLE
- **bpk10Bit:** 10-bit packing mode, the following are possible settings defined in vportcap.h:
  - VPORTCAP_BPK_10BIT_ZERO_EXTENDED
  - VPORTCAP_BPK_10BIT_SIGN_EXTENDED
  - VPORTCAP_BPK_10BIT_DENSE
- **hCtRst:** horizontal counter reset mode. The following are possible settings defined in vportcap.h:
  - VPORTCAP_HRST_EAV
  - VPORTCAP_HRST_START_HBLK
  - VPORTCAP_HRST_START_HSYNC
  - VPORTCAP_HRST_SAV
  - VPORTCAP_HRST_END_HBLK
  - VPORTCAP_HRST_END_HSYNC

- **vCtRst:** vertical counter reset mode. The following are possible settings defined in vportcap.h:
  - VPORTCAP_VRST_START_VBLK
  - VPORTCAP_VRST_START_VSYNC
  - VPORTCAP_VRST_END_VBLK
  - VPORTCAP_VRST_END_VSYNC
  - VPORTCAP_VRST_EAV_V1
  - VPROTCAP_VRST_EAV_V0
- **fldDect:** field detection enable, only used in external sync mode. Indicate whether to use h-sync and v-sync relationship or just use the Field ID input for field detection. The following are possible settings defined in vportcap.h
  - VPORTCAP_FLDD_ENABLE
  - VPORTCAP_FLDD_ENABLE
- **extCtl:** external sync mode enable. The following are possible settings defined in vportcap.h:
  - VPORTCAP_EXC_DISABLE
  - VPORTCAP_EXC_ENABLE
- **fldInv:** field inverse enable. The following are possible settings defined in vportcap.h:
  - VPORTCAP_FINV_DISABLE
  - VPORTCAP_FINV_ENABLE
- **fldXStrt1:** starting pixel number of field one, must be greater than or equal to zero
- **fldYStrt1:** starting line number of field one, must be greater than or equal to one
- **fldXStrt2:** starting pixel number of field two, must be greater than or equal to zero
- **fldYStrt2:** starting line number of field two, must be greater than or equal to one
- **fldXStop1:** the last captured pixel of field one, must be greater than or equal to fldXStrt1
- **fldYStop1:** the last captured line of field one, must be greater than or equal to fldYStrt1
- **fldXStop2:** the last captured pixel of field two, must be greater than or equal to fldXStrt2
- **fldYStop2:** the last captured line of field two, must be greater than or equal to fldYStrt2
- **thrld:** specifies number of double-words required to generate DMA events
- **numFrmBufs:** number of frame buffers to be allocated by driver to store video data captured
- **alignment:** memory alignment requirement for frame buffers
- **mergeFlds:** indicate whether to merge field one and two or to keep them separate in memory. Possible settings are defined in vport.h:
  - VPORT_FLDS_MERGED
  - VPORT_FLDS_SEPARATED

- **SegId:** DSP/BIOS memory segment ID, used by driver for frame buffer allocation
- **edmaPri:** priority level of EDMA transfers
- **irqId:** EDMA interrupt ID

2.2.3 Display Channel Parameters

```
typedef struct {
 Int        dmode;
 Int        fldOp;
 Int        scale;
 Int        resmpl;
 Int        defValEn;
 Int        bpk10Bit;
 Int        vctl1Config;
 Int        vctl2Config;
 Int        vctl3Config;
 Int        extCtl;

 Uint16     frmHSize;
 Uint16     frmVSize;

 Int16      imgHOffsetFld1;
 Int16      imgVOffsetFld1;
 Uint16     imgHSizeFld1;
 Uint16     imgVSizeFld1;

 Int16      imgHOffsetFld2;
 Int16      imgVOffsetFld2;
 Uint16     imgHSizeFld2;
 Uint16     imgVSizeFld2;

 Uint16     hBlnkStart;
 Uint16     hBlnkStop;

 Uint16     vBlnkXStartFld1;
 Uint16     vBlnkYStartFld1;
 Uint16     vBlnkXStopFld1;
 Uint16     vBlnkYStopFld1;
 Uint16     vBlnkXStartFld2;
```

```
    Uint16      vBlnkYStartFld2;
    Uint16      vBlnkXStopFld2;
    Uint16      vBlnkYStopFld2;

    Uint16      xStartFld1;
    Uint16      yStartFld1;
    Uint16      xStartFld2;
    Uint16      yStartFld2;

    Uint16      hSyncStart;
    Uint16      hSyncStop;

    Uint16      vSyncXStartFld1;
    Uint16      vSyncYStartFld1;
    Uint16      vSyncXStopFld1;
    Uint16      vSyncYStopFld1;

    Uint16      vSyncXStartFld2;
    Uint16      vSyncYStartFld2;
    Uint16      vSyncXStopFld2;
    Uint16      vSyncYStopFld2;

    Uint8       yClipLow;
    Uint8       yClipHigh;

    Uint8       cClipLow;
    Uint8       cClipHigh;

    Uint8       yDefVal;
    Uint8       cbDefVal;
    Uint8       crDefVal;

    Int         rgbX;
    Int         incPix;

    Uint16      thrld;

    Int         numFrmBufs;
```

```
    Int         alignment;
    Int         mergeFlds;
    Int         segId;
    Int         edmaPri;
    Int       irqId;
  }VPORTDIS_Params;
```

- **dmode:** display mode, The following are possible settings defined in vport.h:
  - VPORT_MODE_BT656_8BIT
  - VPORT_MODE_BT656_10BIT
  - VPORT_MODE_RAW_8BIT
  - VPORT_MODE_RAW_10BIT
  - VPORT_MODE_YC_8BIT
  - VPORT_MODE_YC_10BIT
  - VPORT_MODE_RAW_16BIT
  - VPORT_MODE_RAW_20BIT
- **fldOp**: field and frame operation mode. The following are possible settings defined in vport.h:
  - VPORT_FLDOP_FLD1
  - VPORT_FLDOP_FLD2
  - VPORT_FLDOP_FRAME
  - VPORT_FLDOP_PROGRESSIVE
- **scale:** horizontal 2x scaling enable, The following are possible settings defined in vport.h:
  - VPORT_SCALING_DISABLE
  - VPORT_SCALING_ENABLE
- **resmpl:** chroma horizontal 4:2:0 to 4:2:2 re-sampling enable. The following are possible settings defined in vport.h:
  - VPORT_RESMPL_DISABLE
  - VPORT_RESMPL_DISABLE
- **defValEn:** default value output enable. Enable output of default value in the non-blanking period outside the image window. The following are possible values defined in vportdis.h:
  - VPORTDIS_DEFVAL_DISABLE
  - VPORTDIS_DEFVAL_ENABLE
- **bpk10Bit:** 10-bit packing mode. The following are possible settings defined in vportdis.h:
  - VPORTDIS_BPK_10BIT_NORMAL
  - VPORTDIS_BPK_10BIT_DENSE

- • **vctl1Config:** VCTL1 pin output select. The following are possible settings defined in vportdis.h:
  - VPORTDIS_VCTL1_HSYNC
  - VPORTDIS_VCTL1_HBLNK
  - VPORTDIS_VCTL1_AVID
  - VPORTDIS_VCTL1_FLD
- **vctl2Config:** VCTL2 pin output select. The following are possible settings defined in vportdis.h:
  - VPORTDIS_VCTL2_VSYNC
  - VPORTDIS_VCTL2_VBLNK
  - VPORTDIS_VCTL2_CSYNC
  - VPORTDIS_VCTL2_FLD
- **vctl3Config:** VCTL3 pin output select. The following are possible settings defined in vportdis.h:
  - VPORTDIS_VCTL3_CBLNK
  - VPORTDIS_VCTL3_FLD
- **extCtl:** external control enable. Indicate whether the video port is operated in master mode, where VCTLx pins are output signals, or in slave mode, where those pins are inputs generated by an external master timing device, such as a video encoder. The following are possible values defined in vportdis.h:
  - VPORTDIS_EXC_DISABLE
  - VPORTDIS_EXC_ENABLE
- **frmHSize:** horizontal size of the video frame, including the blanking period
- **frmVSize:** vertical size of the video frame, including the blanking period
- **imgHOffsetFld1:** display image horizontal offset in Field 1, relative to the end of horizontal blanking. Can be negative to enable output during horizontal blanking.
- **imgVOffsetFld1:** display image vertical offset in Field 1, relative to the end of vertical blanking. Can be negative to enable output during vertical blanking
- **imgHSizeFld1:** display image width in pixels in Field 1
- **imgVSizeFld1:** display image height in lines in Field 1
- **imgHOffsetFld2:** display image horizontal offset in Field 2, relative to the end of horizontal blanking. Can be negative to enable output during horizontal blanking
- **imgVOffsetFld2:** display image vertical offset in Field 2, relative to the end of vertical blanking. Can be negative to enable output during vertical blanking.
- **imgHSizeFld2:** display image width in pixels in Field 2
- **imgVSizeFld2:** display image height in lines in Field 2
- **hBlnkStart:** specifies the pixel number within the line on which horizontal blanking starts

- **hBlnkStop:** specifies the pixel number within the line on which horizontal blanking ends
- **vBlnkXStartFld1:** specifies the pixel number on which the vertical blanking starts for Field 1
- **vBlnkYStartFld1:** specifies the line number on which the vertical blanking starts for Field 1
- **vBlnkXStopFld1:** specifies the pixel number on which the vertical blanking ends for Field 1
- **vblnkYStopFld1:** specifies the line number on which the vertical blanking ends for Field 1
- **vBlnkXStartFld2:** specifies the pixel number on which the vertical blanking starts for Field 2
- **vBlnkYStartFld2:** specifies the line number on which the vertical blanking starts for Field 2
- **vBlnkXStopFld2:** specifies the pixel number on which the vertical blanking ends for Field 2
- **vblnkYStopFld2:** specifies the line number on which the vertical blanking ends for Field 2
- **xStartFld1:** specifies the pixel number on the first line of Field 1 on which the FLD output is de-asserted
- **yStartFld1:** specifies the line number of Field 1 on which the FLD output is de-asserted
- **xStartFld2:** specifies the pixel number on the first line of Field 2 on which the FLD output is asserted
- **yStartFld2:** specifies the line number of Field 2 on which the FLD output is asserted
- **hSyncStart:** specifies the pixel number on which HSYNC is asserted
- **hSyncStop:** specifies the pixel number on which HSYNC is de-asserted
- **vSyncXStartFld1:** specifies the pixel number on which VSYNC is asserted in Field 1
- **vSyncYStartFld1:** specifies the line number on which VSYNC is asserted in Field 1
- **vSyncXStopFld1:** specifies the pixel number on which VSYNC is de-asserted in Field 1
- **vSyncYStopFld1:** specifies the line number on which VSYNC is de-asserted in Field 1
- **vSyncXStartFld2:** specifies the pixel number on which VSYNC is asserted in Field 2
- **vSyncYStartFld2:** specifies the line number on which VSYNC is asserted in Field 2
- **vSyncXStopFld2:** specifies the pixel number on which VSYNC is de-asserted in Field 2
- **vSyncYStopFld1:** specifies the line number on which VSYNC is de-asserted in Field 2
- **yClipLow:** specifies the lower boundary of allowable Y value without clipping
- **yClipHigh:** specifies the upper boundary of allowable Y value without clipping
- **cClipLow:** specifies the lower boundary of allowable Cb/Cr value without clipping
- **cClipHigh:** specifies the upper boundary of allowable Cb/Cr value without clipping
- **yDefVal:** specifies 8 MS bits of the default Y value
- **cbDefVal:** specifies 8 MS bits of the default Cb value
- **crDefVal:** specifies 8 MS bits of the default Cr value
- **rgbX:** only used in raw mode for sequential 24/30-bit RGB data output, RGB extract enable, performing ¾ FIFO unpacking. Please refer to SPRU629 for more information.
- **incPix:** only used in raw mode, specifies that the internal FPCOUNT is incremented every incPix output clocks. For example, incPix would be set to 1 for 16-bit RGB output when used

in 16-bit raw mode, while incPix would be set to 3 for 24-bit RGB output when used in 8-bit raw mode. Please refer to SPRU629 for more information.

- **thrId:** specifies number of double-words required to generate DMA events
- **numFrmBufs:** number of frame buffers to be allocated by driver to store video data captured
- **alignment:** memory alignment requirement for frame buffers
- **mergeFlds:** indicate whether to merge field one and two or to keep them separate in memory. Possible settings are defined in vport.h:
  - VPORT_FLDS_MERGED
  - VPORT_FLDS_SEPARATED
- **SegId:** DSP/BIOS memory segment ID, used by driver for frame buffer allocation
- **edmaPri:** priority level of EDMA transfers
- **irqId:** EDMA interrupt ID

2.2.4 Video Port Global Interrupt Processing

```
typedef struct VPORT_VIntCbParams{
 Int cbArg;
 VPORT_IntCallBack vIntCbFxn;
 Uint16 vIntMask;
 Uint16 vIntLine;
 Int irqId;
 Uns intrMask;
} VPORT_VIntCbParams;
```

- **cbArg:** specifies the argument of the interrupt call-back function
- **vIntCBFxn:** specifies the pointer of the interrupt call-back function
- **vIntMask:** specifies event or events that are enabled to generate the video port global interrupt. The following are possible values defined in vport.h:
  - VPORT_INT_COVR: capture FIFO over-run
  - VPORT_INT_CCMP: capture complete
  - VPORT_INT_SERR: synchronization error
  - VPORT_INT_VINT1: vertical interrupt in field 1
  - VPORT_INT_VINT2: vertical interrupt in field 2
  - VPORT_INT_SFD: short field detected
  - VPORT_INT_LFD: long field detected
  - VPORT_INT_STC: system time clock
  - VPORT_INT_TICK: clock tick
  - VPORT_INT_DUND: display FIFO under-run

  - VPORT_INT_DCMP: display complete
  - VPORT_INT_DCNA: display complete not acknowledged
- **irqId:** specifies the interrupt channel to be used for the video port global interrupt
- **IntrMask:** interupt mask, set while executing ISR

2.2.5 Commands

The following are implemented run-time commands defined in vport.h:

- **VPORT_CMD_RESET:** resets video port
- **VPORT_CMD_CONFIG_PORT:** configures video port
- **VPORT_CMD_CONFIG_CHAN:** configures a video channel
- **VPORT_CMD_START:** starts capture or display operation
- **VPORT_CMD_STOP:** stops capture or display operation
- **VPORT_CMD_SET_VINTCB:** setup video port global interrupt call-back
- **VPORT_CMD_DUND_RECOVER:** force video port to recover from display under-run
- **VPORT_CMD_COVR_RECOVER:** force video port to recover from capture over-run

2.3 Device Parameters for Board Specific Part of the Drivers

2.3.1 The External Device Control (EDC) Interface

As showed in Figure 2, the capture and display mini-drivers consist of the following two parts:

- The generic part, which is designed to work as is with different external video codecs and board layouts. All of its dependencies lies within the DM642 device. For example, the vportdis.c file in the display mini-driver library, can be reused without any code change with different video encoders on different customer boards.
- The board specific part, which only works with specific video encoders or decoders on a specific board. For example, the SAA7105.c file in the display mini-driver library, only works with the Phillips SAA7105 video encoder on the DM642 EVM. Source code changes may be needed even if the same encoder is used on a different board.

The EDC interface is defined to allow seamless integration of these two parts in order to maximize code reuse and minimize possible errors in the integration process. It defines a set of APIs that a board-specific part of the mini-driver must implement in order to work with the generic part.

```
/* EDC control commands */
#define EDC_CONFIG          0x00000001
#define EDC_RESET           0x00000002
#define EDC_START           0x00000003
#define EDC_STOP            0x00000004
#define EDC_GET_CONFIG      0x00000005
#define EDC_GET_STATUS      0x00000006
/* base of user defined commands */
#define EDC_USER            0x10000000
/* EDC return codes */
#define EDC_SUCCESS             0
#define EDC_FAILED              -1
typedef void*  EDC_Handle;
/*
 *  ======== EDC_Fxns ========
 *      edcOpen()       required, open the device
 *      edcClose()      required, close the device
 *      edcCtrl()       required, control/query device
 */
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;
```

2.3.2 SAA7105 Parameters

```
typedef struct {
 SAA7105_AnalogFormat aFmt;
 SAA7105_Mode         mode;
 SAA7105_InputFormat  iFmt;
 Bool                 enableSlaveMode;
 Bool                 enableBT656Sync;
 I2C_Handle           hI2C;
} SAA7105_ConfParams;
```

- **aFmt: s**pecified the analog output format of the video encoder device. Possible values are defined in saa7105.h:

```
typedef enum SAA7105_AnalogFormat {
 SAA7105_AFMT_SVIDEO = 0,
 SAA7105_AFMT_RGB = 1,
 SAA7105_AFMT_YPBPR = 1,
 SAA7105_AFMT_COMPOSITE = 2
} SAA7105_AnalogFormat;
```

- **mode:** specifies the video format. Possible values are defined in saa7105.h:

```
typedef enum
{
 SAA7105_MODE_NTSC720,
 SAA7105_MODE_PAL720,
      SAA7105_MODE_QVGA,
      SAA7105_MODE_VGA,
      SAA7105_MODE_SVGA,
      SAA7105_MODE_XGA,
      SAA7105_MODE_HD480P60F,
      SAA7105_MODE_HD720P24F,
      SAA7105_MODE_HD720P60F,
      SAA7105_MODE_HD1080I30F
}SAA7105_Mode;
```

- **iFmt:** input format. Possible values are defined in saa7105.h:

```
typedef enum SAA7105_InputFormat {
    SAA7105_IFMT_RGB24_YCBCR444,
    SAA7105_IFMT_RGB555,
    SAA7105_IFMT_RGB565,
    SAA7105_IFMT_YCBCR422_NONEINTERLACED,
    SAA7105_IFMT_YCBCR422_INTERLACED
}SAA7105_InputFormat;
```

- **enableSlaveMode:** specifies whether device is operated in master or slave mode: Possible values are :
  - 1, for slave mode
  - 0, for master mode

Note: because the design of the DM642 EVM board requires that the video port 2 work in master mode, SAA7105 must then work in slave mode to ensure proper operation.

- **enableBT656Sync:** enable embedded synchronization using SAV/EAV code defined in ITU-R BT.656.
- **hI2C:** handle to the DM642 I2C controller

2.3.3 SAA7115 Parameters

```
typedef struct {
 SAA7115_Mode inMode;
 SAA7115_Mode outMode;
 SAA7115_AnalogFormat aFmt;
 Bool enableBT656Sync;
 Bool enableIPortOutput;
 I2C_Handle hI2C;
 Int hSize;
 Int vSize;
 Bool interlaced;
} SAA7115_ConfParams;
```

- **inMode: s**pecifies the input video format. Possible values are defined in saa7115.h:

```
typedef enum SAA7115_Mode{
 SAA7115_MODE_NTSC640,
 SAA7115_MODE_NTSC720,
 SAA7115_MODE_PAL720,
 SAA7115_MODE_PAL768,
 SAA7115_MODE_CIF,
 SAA7115_MODE_QCIF,
 SAA7115_MODE_SQCIF,
 SAA7115_MODE_SIF
 SAA7115_MODE_USER
}SAA7115_Mode;
```

- **outMode:** specifies the output video format. Possible values are the same as inMode described above. Since SAA7115 has an on-chip scaling capabilities. output format can be different from input format.
- **aFmt:** specified the analog output format of the video encoder device. Possible values are defined in saa7105.h:

```
typedef enum SAA7115_AnalogFormat {
    SAA7115_AFMT_SVIDEO,
    SAA7115_AFMT_COMPOSITE
} SAA7115_AnalogFormat;
```

- **enableBT656Sync:** enable insertion of SAV/EAV code defined in ITU-R BT.656 into the output video data stream
- **enableIPortOutput**: enable video data output from the I-PORT instead of the X-PORT
- **hI2C:** handle to the DM642 I2C controller

The following parameters are used optionally when inMode = SAA7115_MODE_USER:

- **hSize:** horizontal size of the user-defined image
- **vSize:** vertical size of the user-defined image
- **interlaced:** specify whether user-defined image is in the interlaced or progressive format

2.3.4 String Naming Convention in FVID_create()

Since the DM642 video port is capable of dual-channel capture operations, and since the generic part of the driver can be hooked up with any EDC compliant module for external codec configuration, there must be a way for this information to be passed to the driver from the application for the driver to function properly. This is done when calling FVID_create(). The first argument of the function is ***name***, which is of type ***String***. The following rules are applied to the definition of that string.

- For the Capture Driver: the string consists of up to 3 sub-strings, separated by '/'. For example, "VP0CAPTURE/A/0".

- The first sub-string shall be the name of the driver defined in the DSP/BIOS .cdb file. This sub-string and its associated device ID are used by the GIO class driver to identify the video port.
- The second sub-string shall be either A or B to identify whether this is for video port channel A or channel B.
- The third sub-string is optional, and if specified, is used to identify the external video codec. For example, on the DM642 EVM board, there are two SAA7115s that use the same EDC compliant module to get initialized. The EDC module must know which codec to initialize and configure. This is done by make the third sub-string either “0” or “1”.
- The naming convention of the string is similar for the display driver, except that it only consists of up to two sub-strings because it always operates in single-channel mode. For example, “VP2DISPLAY”.
- The first sub-string shall be the name of the driver defined in the DSP/BIOS .CDB configuration file. This sub-string and its associated device id are used by the GIO class driver to identify the video port.
- The second sub-string is optional, and if specified, is used to identify the external video codec. For the DM642 EVM board, this sub-string is ignored since there is only one SAA7105 device.

3 Architecture

This section describes the design and implementation of the device driver. The driver uses various DSP/BIOS and CSL modules (see Appendix A). Refer to the *TMS320C6000 DSP/BIOS Application Programming Interface* (literature number SPRU403) and *TMS320C6000 Chip Support Library API Reference Guide* (literature number SPRU401). The technical details of the EDMA are available from *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190). The technical details of the video port are available from TMS320C64x DSP Video Port/ VCXO Interpolated Control (VIC) Port Reference Guide (literature number SPRU629).

3.1 Block Diagram

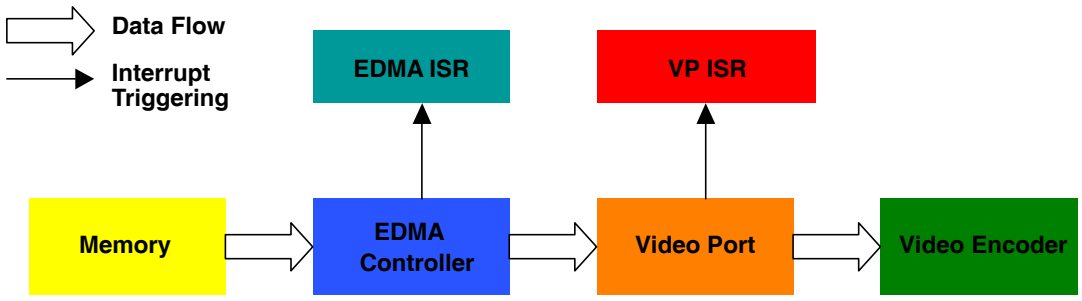

Figure 3. Block Diagram of the Display Driver

Figure 3 shows the top-level block diagram of the display driver. In display operation, data flows from frame buffer located in memory to the video port FIFO by EDMA transfers. The video port in turn outputs data to the external video encoder for display. EDMA interrupt is triggered after an entire frame is transferred from memory to the video port. This interrupt is essential and is used by the driver for the following purposes:

- Frame buffer management
- EDMA re-load entry updating
- Notifying the class driver that an empty frame buffer is available for the application to fill by calling the callback function provided by the class driver at initialization time
- The video port global interrupt can be optionally enabled by the application for error handling or for synchronization with the video port.

The capture driver is very similar except that data flows in opposite direction.

3.2 Buffer Management

Frame buffers containing video data are allocated and initially owned by the drivers. The number of frame buffers that the drivers allocate is run-time configurable with a minimum requirement of triple buffering. Before allocation, the 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 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_free() 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.

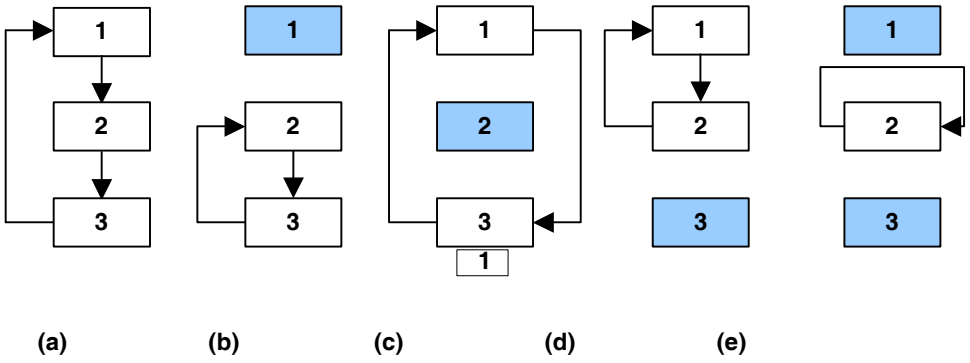

Figure 4. Capture Driver Buffer Management

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) (e).

When the application calls FVID_free(), 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() and FVID_alloc() sequentially, as shown in Figure 4 from (b) to (c) and from (c) to (d).

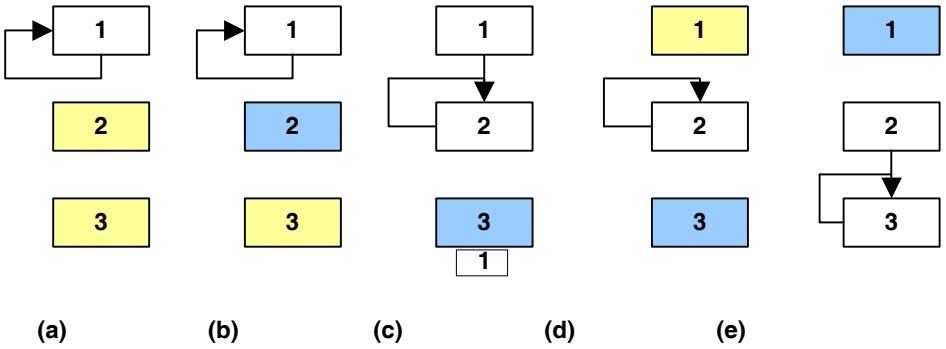

Figure 5. Display Driver Buffer Management.

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(), it gets a buffer from the driver and 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(), it returns a buffer ready for display back to the driver. The driver, in turn, will set this buffer as its now ***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() and FVID_alloc() sequentially, as shown in Figure 5(d) to (e).

3.3 Cache Coherency

It is the application's responsibility to ensure cache coherency, as the driver does nothing in this respect. This is because data is typically moved by EDMA between fast on-chip SRAM 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.

4 Constraints

This device driver does not support the following modes:

- Raw data capture
- Dual-channel synchronized raw-data display
- Synchronized to another video port
- Capture of MPEG transport stream

5 References

All these documents are available from *http://www.ti.com*.

1. *DSP/BIOS Driver Developer's Guide* (SPRU616)
2. *TMS320C64x DSP Video Port/ VCXO Interpolated Control (VIC) Port Reference Guide* (SPRU629)
3. *TMS320C6000 DSP/BIOS Application Programming Interface* (SPRU403)
4. *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401)
5. *TMS320C6000 Peripherals Reference Guide* (SPRU190)

Appendix A Device Driver Data Sheet

A.1 Device Driver Library Name

vportcap_.l64 and vportdis_.l64 for TMS320DM642 device.

A.2 DSP/BIOS Modules Used

- HWI – Hardware Interrupt Manager
- QUE – Queue Manager
- IOM – I/O Manager

A.3 DSP/BIOS Objects Used

- QUE_Obj

A.4 CSL Modules Used

- EDMA module
- IRQ module
- I2C module
- VP module

A.5 CPU Interrupts Used

- EDMA interrupt
- Video Port interrupt

A.6 Peripherals Used

- EDMA
- EMIF
- I2C
- Video Port

A.7 Maximum Interrupt Latency(Capture/Display)

89/71 cycles

A.8 Memory Usage

Table A–1. Device Driver Memory Usage (Capture/Display)

	Uninitialized memory	Initialized memory
CODE	–––	16704/18240 Words
DATA	3968/4186 + memory dynamically allocated for video frame buffers	156/384 Words

Appendix B The FVID API Interface

B.1 Overview

The FVID module presents an API for DSP/BIOS applications that are performing 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.

The FVID device driver API differs from many other drivers in the way that it manages ownership of data buffers with the application. Most DSP/BIOS device drivers expect the application to initially have control of data buffers and to pass the address of these buffers down to the device driver for the purposes of data input or output. The FVID module and the video capture and display mini-drivers that it works with use an inverted model in which the device driver assumes initial control of the data buffers. The motivation for this approach is that video systems often have large, dedicated memory buffers that are optimized for high-speed data movement, rather than being allocated from the system memory heap. Since the size, location, and status of these buffers are set within the driver itself, it makes sense that the driver would have overall control of them.

Frame video systems are also characterized by efficient multi-buffering of frame data. Double buffering is used so that when one buffer is being used to move data to or from the display hardware (and therefore owned by the driver), the other buffer (is owned by and) is being used by the application for rendering. The two buffers can then be swapped during a frame-sync and the process will be repeated. Triple buffering is often used to allow the application to obtain a buffer from the driver without having to wait for a new frame-sync.

B.2 Using the FVID APIs

As discussed above, the device driver has initial ownership of the frame buffers and the application needs to allocate these buffers for processing. Therefore, the prime role of the FVID APIs is managing ownership of the buffers between the device driver and the application. At the heart of the FVID interface are two calls:

FVID_alloc. Allocate a video port buffer to the application.

FVID_exchange. Exchange an application-owned buffer for a driver-owned buffer.

Once the driver is initialized, the application will need to call FVID_alloc once at the beginning to get initial ownership of the buffer. After that, calls to FVID_exchange can be used to swap buffers between the application and driver.

The FVID interface is completely integrated into DSP/BIOS. For each FVID channel created, a separate synchronization object is initialized. Calls to allocate and exchange video buffers can be blocking or non-blocking, depending on the timeout value specified when the channel was created.

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

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

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

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

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

    /* create and initialize the FVID channel objects */
    capChan = FVID_create("/vcap", IOM_INPUT, NULL, (Ptr)&capParams, NULL);
    disChan = FVID_create("/vdis", IOM_OUTPUT, NULL, (Ptr)&disParams, NULL);

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

    while(1) {
        /* copy captured frame data to the display frame buffer */
        FrameDataCopy(capFrameBuf, disFrameBuf);

        FVID_exchange(capChan, &capFrameBuf);
        FVID_exchange(disChan, &disFrameBuf);
    }
}
```

B.3 API Description

B.3.1 Functions

The following API functions are defined by the FVID module:

- FVID_alloc. Allocate a video port buffer to the application.
- FVID_control. Send a control command to the mini-driver.
- FVID_create. Allocate and initialize an FVID channel object.
- FVID_delete. De-allocate an FVID channel object.
- FVID_exchange. Exchange an application-owned buffer for a driver-owned buffer.
- FVID_free. Relinquish a video port buffer back to the driver.

B.3.2 Constants, Types, and Structures

```
/* definition of interlaced frame */
typedef struct FVID_IFrame{
    unsigned char* y1;
    unsigned char* cb1;
    unsigned char* cr1;
    unsigned char* y2;
    unsigned char* cb2;
    unsigned char* cr2;
}FVID_IFrame;

/* progressive frame */
typedef struct FVID_PFrame {
    unsigned char* y;
    unsigned char* cb;
    unsigned char* cr;
} FVID_PFrame;

/* Raw frame, could be RGB, monochrome or just any data*/
/* interleaved Y/C frame etc. */
typedef struct FVID_RawIFrame{
    unsigned char* buf1;
    unsigned char* buf2;
} FVID_RawIFrame;

typedef struct FVID_RawPFrame{
    unsigned char* buf;
} FVID_RawPFrame;

/* definition of interlaced frame */
typedef struct FVID_IFrame{
    unsigned char* y1;
    unsigned char* cb1;
    unsigned char* cr1;
    unsigned char* y2;
    unsigned char* cb2;
    unsigned char* cr2;
}FVID_IFrame;

/* progressive frame */
typedef struct FVID_PFrame {
    unsigned char* y;
    unsigned char* cb;
    unsigned char* cr;
} FVID_PFrame;

/* Raw frame, could be RGB, monochrome or just any data*/
/* interleaved Y/C frame etc. */
typedef struct FVID_RawIFrame{
```

```
        unsigned char* buf1;
        unsigned char* buf2;
    } FVID_RawIFrame;

    typedef struct FVID_RawPFrame{
        unsigned char* buf;
    } FVID_RawPFrame;
    /* FVID frame buffer descriptor */
    typedef struct FVID_Frame {
        QUE_Elem        queElement; /* the first two words are for queuing   */
        union {
            FVID_IFrame     iFrm;     /* y/c frame buffer         */
            FVID_PFrame     pFrm;     /* y/c frame buffer         */
            FVID_RawIFrame  riFrm;    /* raw frame buffer         */
            FVID_RawPFrame  rpFrm;    /* raw frame buffer         */
        } frame;
          } FVID_Frame;
```

B.4 Function Calls

FVID_alloc Allocate a video port buffer to the application

Syntax status = FVID_alloc (fvidChan, bufp);

Parameters

FVID_Handle fvidChan	/* handle to an instance of the device */
Ptr bufp	/* pointer to buffer allocated by driver */

Return Value

Int status	/* returns IOM_COMPLETED if successful */

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

The fvidChan argument is the handle of the video driver channel that was created with a call to FVID_create.

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.

FVID_alloc returns IOM_COMPLETED when it completes successfully. If the request is queued by the mini-driver, a status of IOM_PENDING is returned. If an error occurs, a negative value will be returned. See the iom.h header file for the possible error values that can be returned.

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

Example

```
/* allocate a buffer from the device */
status = FVID_alloc(chanHandle, dispBuf);
```

FVID_control Send a control command to the mini-driver

Syntax status = FVID_control (fvidChan, cmd, args);

Parameters

```
FVID_Handle  fvidChan    /* handle to an instance of the device */
Int cmd                  /* control command */
Ptr args                 /* pointer to control command arguments */
```

Return Value

```
Int status               /* returns IOM_COMPLETED if successful */
```

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

The fvidChan argument is the handle of the video driver channel that was created with a call to FVID_create.

The cmd argument specified the control command. At present, the video port mini-driver implements the following control commands.

Table B–1. Device Driver Control Commands

Command	Arg	Function
IOM_ABORT	N/A	Abort all pending I/O jobs
IOM_FLUSH	N/A	All video capture I/O jobs pending are discarded. All video display I/O jobs are processed normally.

The args argument is a pointer to the argument or structure of arguments that are specific to the command being passed

FVID_control returns IOM_COMPLETED when it completes successfully. If an error occurs, this call will return a negative value. See the iom.h header file for the possible error values that can be returned.

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

Example

```
/* abort all pending video driver I/O jobs */
FVID_control(fvidChan, IOM_ABORT, NULL);
```

FVID_create Allocate and initialize an FVID channel object

Syntax fvidChan = FVID_create (name, mode, *status, optArgs, *attrs);

Parameters

String name	/* handle to an instance of the device */
Int mode	/* pointer to buffer allocated by driver */
Int *status	/* pointer to size of buffer pointed to by */
Ptr optArgs	/* */
FVID_Attrs *attrs	/* */

Return Value FVID_Handle fvidChan /* handle to an instance of the device */

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

The name argument is the name specified for the device when it was created in the configuration file or at run-time.

The mode argument specifies the mode in which the device is to be opened. It can be either IOM_INPUT for video capture or IOM_OUTPUT for video display.

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

The attrs argument is a pointer to a structure of type FVID_Attrs:

```
typedef struct FVID_Attrs {
    Uns timeout;
} FVID_Attrs;
```

The timeout member of the attributes structure is to specify the timeout of the synchronization (semaphore) object that is created by the class driver. A value of SYS_FOREVER will cause the FVID_alloc, FVID_free and FVID_exchange calls to wait indefinitely for a completion of the call. A numerical timeout value will cause these APIs to block for the specified time, in units of system clock ticks. A value of 0 will cause the APIs to be non-blocking and they will return immediately, in which case the application would have to check the status returned to make sure the call was completed successfully. If a non-zero timeout is specified, it is important that calls to FVID_alloc, FVID_free and FVID_exchange are only called within a DSP/BIOS task (TSK).

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. This function will return NULL if the device channel could not be opened.

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

Example

```
/* Initialize the attributes */
FVID_ATTRS dispAttrs = FVID_ATTRS;
/* Create an instance to a video display device */ chan-
Handle = FVID_create("\display0", IOM_INPUT, NULL, NULL,
&dispAttrs);
```

FVID_delete Deallocate an FVID channel object

Syntax status = FVID_delete (fvidChan);

Parameters FVID_Handle fvidChan /* handle to an instance of the device */

Return Value Int status /* returns IOM_COMPLETED if successful */

Description An application calls FVID_delete to close a device driver channel.

The fvidChan argument is the handle of the video driver channel that was created with a call to FVID_create.

FVID_delete returns IOM_COMPLETED when it completes successfully. If an error occurs, a negative value will be returned. See the iom.h header file for the possible error values that can be returned.

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

Example

```
/* allocate a buffer from the device */

status = FVID_delete(chanHandle);
```

FVID_exchange Exchange an application-owned buffer for a driver-owned buffer

Syntax status = FVID_exchange (fvidChan, bufp);

Parameters

FVID_Handle fvidChan	/* handle to an instance of the device */
Ptr bufp	/* pointer to buffer exchanged by driver */
LgUns *pSize	/* pointer to size of buffer pointed to by bufp */

Return Value Int status /* returns IOM_COMPLETED if successful */

Description

An application calls FVID_exchange to request the video device driver to give it ownership of a data buffer in exchange for a buffer that the application owns and is ready to relinquish back to the driver. A call to FVID_exchange is functionally equivalent to serial calls to FVID_free and FVID_alloc, but allows the same thing to be done in a single API call. The application will need to call FVID_alloc once to initialize ownership of the frame. This API function will result in an mdSubmit call being made to the mini-driver.

The fvidChan argument is the handle of the video driver channel that was created with a call to FVID_create.

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.

FVID_exchange returns IOM_COMPLETED when it completes successfully. If the request is queued by the mini-driver, a status of IOM_PENDING is returned. If an error occurs, a negative value will be returned. See the iom.h header file for the possible error values that can be returned.

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

Example

```
/* allocate a buffer from the device */

status = FVID_exchange(chanHandle, dispBuf);
```

FVID_free Relinquish a video port buffer back to the driver

Syntax status = FVID_free (fvidChan, bufp);

Parameters

FVID_Handle fvidChan	/* handle to an instance of the device */
Ptr bufp	/* pointer to buffer to be relinquished to driver */
LgUns *pSize	/* pointer to size of buffer pointed to by bufp */

Return Value Int status /* returns IOM_COMPLETED if successful */

Description 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.

The fvidChan argument is the handle of the video driver channel that was created with a call to FVID_create.

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.

FVID_alloc returns IOM_COMPLETED when it completes successfully. If the request is queued by the mini-driver, a status of IOM_PENDING is returned. If an error occurs, a negative value will be returned. See the iom.h header file for the possible error values that can be returned.

Constraints This function can only be called after the device has been loaded and initialized. The handle supplied as an argument to this function should have been obtained with a previous call to FVID_create. The pointer that is passed as an argument to this call must point to a video buffer of type FVID_Frame that was already allocated by the driver through a call to FVID_alloc or FVID_exchange.

Example

```
/* free a buffer back to the device */

status = FVID_free(chanHandle, dispBuf);
```