LSP 1.20 DaVinci Linux V4L2 Display Driver

User's Guide

Literature Number: SPRUEL3
March 2008

TEXAS INSTRUMENTS

Contents

1 Introduction **5**

1.1 Overview 5

1.2 Supported Features 5

2 Driver Overview **6**

2.1 New Video Driver Architecture 6

2.2 V4L2 Driver Architecture 9

2.3 Design 10

2.4 Interrupt Handling 19

3 User Interface **20**

3.1 Data-Structures 20

3.2 Enumerations and Defines 23

3.3 API Specification 24

4 Build and Installation **29**

4.1 Build Steps 29

4.2 Boot Arguments 30

4.3 Installation 30

5 Example Applications **31**

List of Figures

1 DM355/DM6446 Display Interface for Video Windows 5
2 DaVinci Video Display Driver Architecture 7
3 Encoder Interface 8
4 DaVinci Display Driver Basic Architecture 9
5 Relationship Between First Interrupt and Incoming Data 19

List of Tables

1 Responsibilities of Encoder Manager and Encoders 9

User's Guide
SPRUEL3–March 2008

LSP 1.20 DaVinci Linux V4L2 Display Driver

This guide provides details concerning the use of the DaVinci Linux V4L2 Display Driver. For LSP 1.20, the V4L2 Display Driver is supported on the following EVMs: DM644x, DM355.

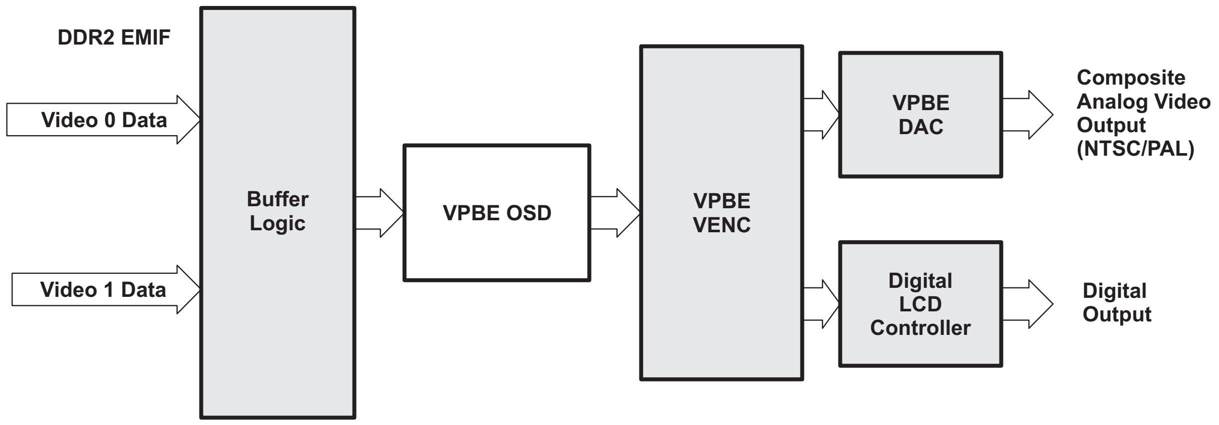
1 Introduction

1.1 Overview

The DaVinci Display Video module is the V4L2 Display driver for DM355/DM6446 VPBE video windows (VID0 and VID1). The module drives one video SD output using video data input from the DDR either independently or as an overlay. For HD, it uses VID0 to drive a component output through the THS8200 daughtercard. In the overlay scenario, Video 1 gets overlaid over the Video 0 window. This guide focuses on the usage of the video output/display driver from an application.

Figure 1 shows the basic block diagram of the DM355/DM6446 VPBE display interface.

Figure 1. DM355/DM6446 Display Interface for Video Windows



The following encoders are used for various types of display:

- **SD:** Built-in VPBE encoder for NTSC and PAL
- **VGA:** Logic PD encoder
- **HD:** THS8200 encoder

The V4L2 Display driver model is the driver considered for the DaVinci Display module. The V4L2 (Video for Linux-2) driver model is a widely used model across many platforms in the Linux community. V4L2 provides a good streaming support with support for a lot of buffer formats. It also has a buffer management mechanism of its own that can be used.

1.2 Supported Features

- The driver can be built as a static module or as a dynamic module.
- Two software channels of display (Video2 and Video3) are supported. Video2 is used for Video window 0 of the VPBE (VIDWIN0) and Video3 is used for Video window 1 of the VPBE (VIDWIN1).
- One software channel (Video2) is available for HD resolutions (720p and 1080i).

- Supports single I/O instance and multiple control instances of each of the channels.
- Supports buffer access mechanism through memory mapping as well as user pointers.

2 Driver Overview

2.1 New Video Driver Architecture

The FBDev and V4L2 drivers implement the lower layers of the corresponding framework. In previous releases, these drivers used separate hardware modules to invoke the lower-layer services of the hardware. Since the underlying hardware is the same and both of these drivers configure the hardware independently, they could not co-exist on the target platform. The new video architecture addressed this issue by abstracting the lower-layer services in a set of common hardware modules and invoking these services from the V4L2 and FBDev drivers. It is anticipated that application will use the FBDev devices for displaying graphics using the OSD layers of the hardware and V4L2 devices for streaming video using the video layers of the hardware. To provide backward compatibility, video layers can still be used by the FBDev driver (but would need to be configured using bootargs), but they will be unavailable for V4L2 driver.

The following are the high level requirements for the architecture:

1. Allow both FBDev and V4L2 to use common hardware modules
2. Can be re-used across multiple platforms with similar hardware model
   - Changes confined to hardware layer
   - Minimum changes to hardware independent layer
3. Re-use the encoder interface developed in DaVinci HD that allows seamless integration of encoders to support multiple video and graphics resolutions.

Based on this, the software functionality is divided into two layers as shown in Figure 2.

1. Hardware-independent layer
2. Hardware-dependent layer

The hardware-independent layer consists of:

1. Frame Buffer Driver (FBDev)
2. V4L2 Driver
3. Sysfs
4. Encoder Manager

The hardware-dependent layer consists of:

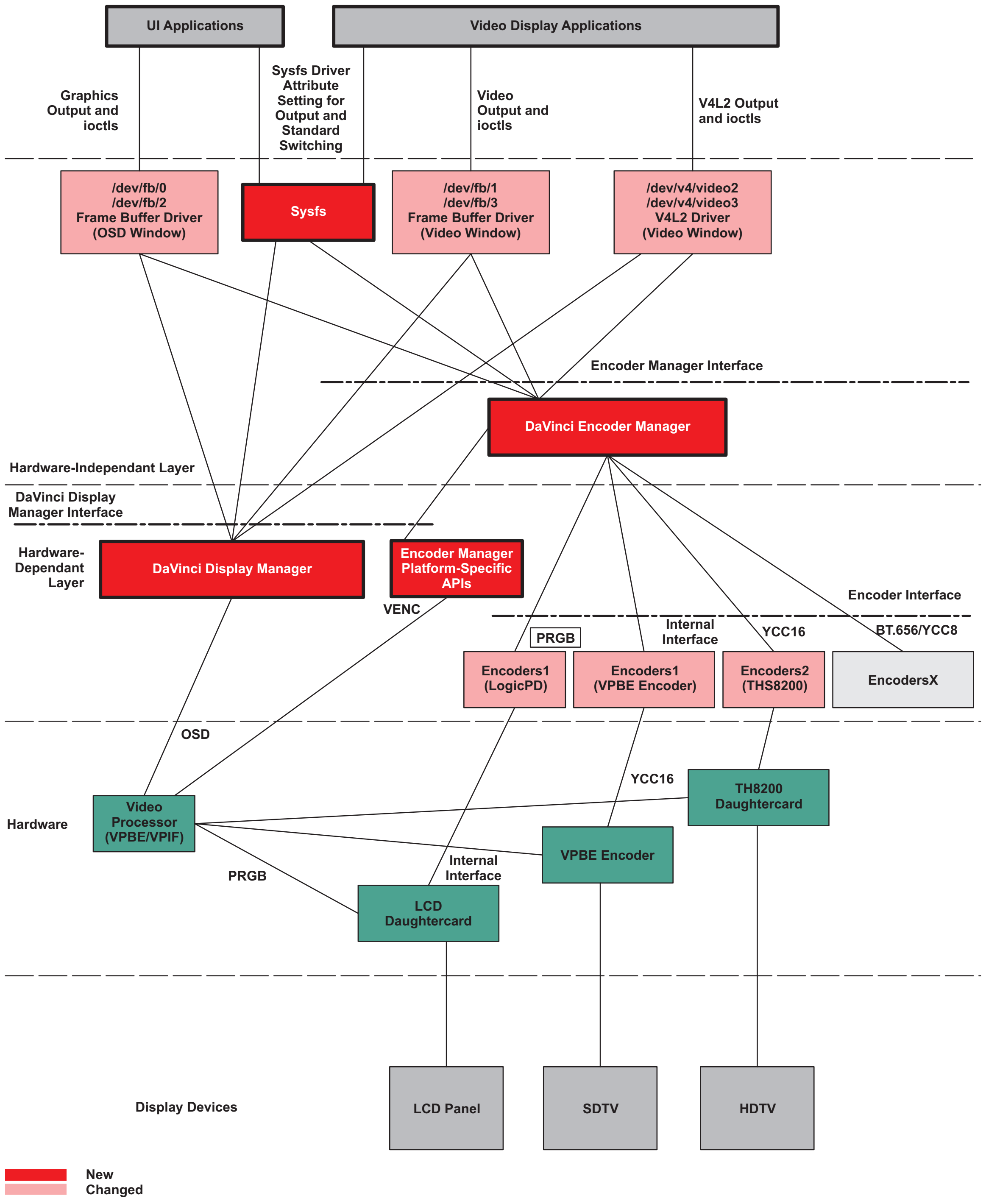
1. DaVinci Display Manager
2. Encoder Manager Platform APIs
3. Encoders

The following color coding is used in the Figure 2:

- PINK - existing modules modified as per new architecture
- RED - new modules added as per new architecture

The existing modules, V4L2 and FBDev, have been modified to remove the hardware-dependent layer functionality. Now the modules use the APIs defined by the hardware-dependent layer, instead. For FBDev, the existing implementation is re-visited to eliminate a few proprietary IOCTLs that are implemented incorrectly and has replaced them with standard APIs. Current V4L2 implementation used an encoder interface that was developed for DaVinci HD for easy integration of hardware encoders to the driver. However, it has been developed with a V4L2 bias. This is re-used in this architecture by eliminating V4L2-specific definitions with standard C definitions. The following sections discuss high-level details of the new modules introduced in this architecture.

Figure 2. DaVinci Video Display Driver Architecture


UI Applications
Video Display Applications
Graphics Output and ioctls
Sysfs Driver Attribute Setting for Output and Standard Switching
Video Output and ioctls
V4L2 Output and ioctls
/dev/fb/0 /dev/fb/2 Frame Buffer Driver (OSD Window)
Sysfs
/dev/fb/1 /dev/fb/3 Frame Buffer Driver (Video Window)
/dev/v4/video2 /dev/v4/video3 V4L2 Driver (Video Window)
Encoder Manager Interface
DaVinci Encoder Manager
Hardware-Independant Layer
DaVinci Display Manager Interface
Hardware-Dependant Layer
DaVinci Display Manager
Encoder Manager Platform-Specific APIs
VENC
Encoder Interface
PRGB
Internal Interface
YCC16
BT.656/YCC8
Encoders1 (LogicPD)
Encoders1 (VPBE Encoder)
Encoders2 (THS8200)
EncodersX
OSD
YCC16
TH8200 Daughtercard
Hardware
Video Processor (VPBE/VPIF)
VPBE Encoder
PRGB
Internal Interface
LCD Daughtercard
Display Devices
LCD Panel
SDTV
HDTV
New
Changed

2.1.1 DaVinci Display Manager

The DaVinci Display Manager for DM355 and DM6446 is responsible for the following functionality:

- Layer management. All OSD and video layers are initially owned by the Display Manager and are allocated to front-end drivers (V4L2 and FBDev), as needed. FBDev claims the OSD layers at initialization and releases them at exit of the driver. V4L2 claims the video layer at run time when the device is opened and releases it at close of the device. FBDev claims the video layers if it is configured through boot arguments. By default (without any boot arguments for video layers), these layers are not claimed by FBDev and will be available for use by V4L2.
- Provides service to FBDev and V4L2 drivers to configure the OSD hardware. This involves setting buffer address, line length, blending, zooming/scaling, window dimension, and other related functionality.
- Color lookup table management. This allows configuration RAM/ROM CLUTs for use and update to RAM CLUT.
- Attribute and cursor settings. When one of the OSD layers is used in bitmap mode, the other OSD layer may be configured as an attribute layer. Also allows setting the cursor position and blinking.
- ISR event reporting. Both drivers schedule the video/graphics buffer for display when this event is received and mark the finished buffers for re-use by the application.
- Other miscellaneous functions, as listed in the OSD sections of the *TMS320DM644x DMSoC Video Processing Back End (VPBE) User's Guide* (SPRUE37) and the *TMS320DM35x Digital Media System-on-Chip (DMSoC) Video Processing Back End Reference Guide* (SPRUF72).

2.1.2 DaVinci Encoder Manager

The DaVinci Encoder Manager is responsible for the following:

- Manages the registration and de-registration of encoders that implement a set of API calls.
- Provides API to allow set/get of output (composite, s-video etc), standard/mode (NTSC, PAL, VGA etc), control (brightness, hue, contrast etc) and parameters.
- Platform-specific functions. Some of the platform needs settings in the VPBE/VPIF to allow configuration of the digital port. This involves formatting the digital port for the required interface type (YCbCr/YCC8/YCC16/BT.656/PRGB/SRGB) and generating timing signals required for the selected standard/mode. These are abstracted as APIs and are implemented by the respective platforms. If a specific platform does not have any such functionality, it implements a dummy API call that does nothing.

The DaVinci Encoder Manager manages a list of encoders. Encoders register with the manager and implement a set of standard API calls that are used to control the operation of the encoder (shown as encoder interface in Figure 2). The manager hides the detail of which encoder is supported, what standard and output is used, and provides a set of generic APIs to invoke its services. Figure 3 shows how the encoders are interfaced to the SoC at the digital video port (shown as interface that could be YCbCr/YCC8/YCC16/BT.656/PRGB/SRGB).

Figure 3. Encoder Interface

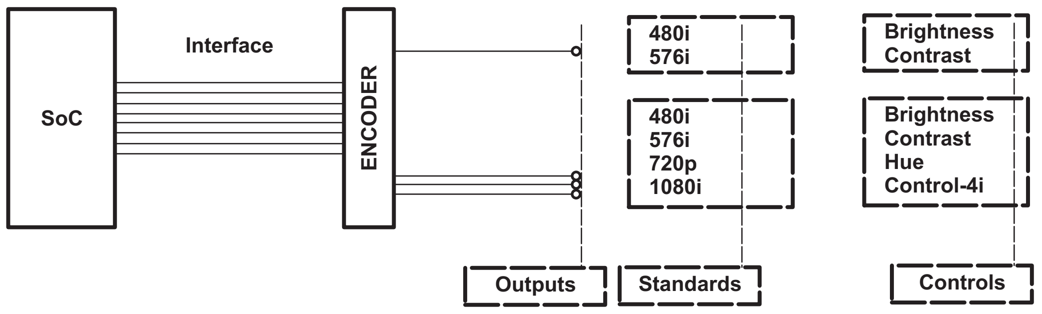


Table 1 lists the responsibilities of encoder manager and encoders for each of the commands it services.

Table 1. Responsibilities of Encoder Manager and Encoders

Command	Encoder Manager	Encoder
Set output	Set current encoder to encoder that supports this output. Call the encoder's API to set the output.	Set requested output and default standard.
Get output	Call current encoder's API to get the output.	Return current output.
Enumerate outputs	Enumerate the outputs supported by the encoder.	Return output name at the given index.
Set Mode	Set the platform digital port as required for the mode. Call current encoder's API to set the mode at the encoder.	Set the requested mode.
Get Mode	Call current encoder's API to get mode.	Return current mode.
Set control	Call current encoder's API to set control.	Set control at current output.
Get control	Call current encoder's API to Get control value.	Return current control value.
Set parameters	Call current encoder's API to set parameters.	Set parameters at the encoder.
Get parameters	Call current encoder's API to get parameters.	Get parameters at the encoder.

All driver modules use the common strings for output name and modes as defined in the **vid_encoder_types.h** header file. The encoder API specification is available in **vid_encoder_if.h**. To add a custom encoder, you need to implement the APIs defined in this header file.

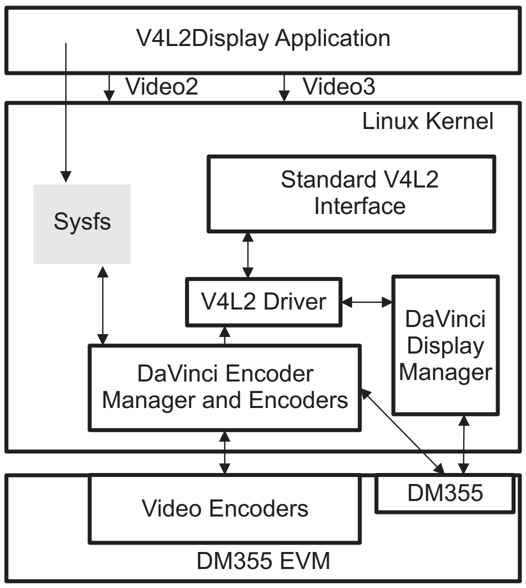
2.1.3 Sysfs

V4L2 specifications allow switching output and standard using IOCTLs. FBDev specifications allow switching of resolutions at the output, but not the output itself. In the past, proprietary IOCTLs were added in FBDev to allow switching of output. Instead of abusing the FBDev interface with proprietary IOCTLs, it was decided to remove this functionality from V4L2 and FBDev and implement the same as a Sysfs driver attribute. This can be extended to support simple functions like enable/disable display, control brightness, hue etc. The *LSP 1.20 DaVinci Video Sysfs User's Guide* (SPRUEL6) explains the procedure to change the output and standard to work with the current display device.

2.2 *V4L2 Driver Architecture*

Figure 4 shows basic architecture of the DaVinci Display Driver. The driver invokes the services of the display manager and encoder manager (discussed in section Section 2.1) modules to implement the lower layer functionality.

Figure 4. DaVinci Display Driver Basic Architecture



The DaVinci Display Driver is based on the V4L2 driver model. The driver can be used to stream video to display hardware supporting the following standards:

- SD output display: NTSC 480i 30 FPS, PAL 576i 25 FPS
- ED output display: NTSC 480p 60 FPS and PAL 576p 50 FPS
- HD output display (through THS8200 daughtercard): 720p 60FPS and 1080i 30FPS
- LCD display: 640x480, 640x400, and 640x350

2.3 Design

2.3.1 Opening and Closing the Driver

The device can be opened using an open call from the application with device name and mode of operation as parameters. The application can open the driver in either blocking mode or non-blocking mode. If the channel is opened in non-blocking mode, the driver calls (call to DQBUFF) return to the application without waiting for request completion.

The driver exposes two software channels (Video2 and Video3), one each for hardware video windows VIDWIN0 and VIDWIN1. Both the software channels support SD display. Only the Video2 channel supports HD display.

The driver supports a single I/O instance and multiple control instances of each of the channels.

The file descriptor, with which VIDIOC_REQBUF ioctl is called, becomes I/O instance. The remainder will be control instances.

The application can call the close function with the file handle to close a specific device. The application should close all the control channels before closing an I/O channel.

```
/* call to open a video Display logical channel in blocking mode */
video2fd_blocking =open ("/dev/video2", O_RDWR);

/* call to open a video Display logical channel in non-blocking mode */
video2fd_nonblocking =open ("/dev/video2", O_RDWR | O_NONBLOCK);

/* closing of channels */
close (video2fd_blocking);
close (video2fd_nonblocking);
```

2.3.2 Buffer Management

The driver allows two different types of memory allocation modes:

- Driver-buffer mode
- User-buffer mode

For driver-buffer mode, the application requests memory from the driver by calling VIDIOC_REQBUFS. In the case of user-buffer mode, the application needs to allocate physically contiguous memory using some other mechanism in user space. In driver-buffer mode, the maximum number of buffers is limited to VIDEO_MAX_FRAME (defined in driver header files).

Below are the major steps the application needs to perform for buffer allocation:

1. **Allocating memory**

   This IOCTL is used to allocate memory for frame buffers.

   Ioctl: VIDIOC_REQBUFS.

   It takes a pointer to the instance of v4l2_requestbuffers structure as an argument. You can specify the buffer type (V4L2_BUF_TYPE_VIDEO_OUTPUT), number of buffers, and memory type (V4L2_MEMORY_MMAP) at the time of buffer allocation. The file descriptor that calls the VIDIOC_REQBUFS ioctl, is considered an I/O instance.

   Constraint: This ioctl can be called only once from the application.

   Example:

   ```
   /* structure to store buffer request parameters */
   struct v4l2_requestbuffers reqbuf;
   ```

```
reqbuf.count = numbuffers;
reqbuf.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
reqbuf.memory = V4L2_MEMORY_MMAP;

ret = ioctl(fd , VIDIOC_REQBUFS, &reqbuf);
if(ret) {
    printf("cannot allocate memory\n");
    close(fd);
    return -1;
}
```

2. **Getting physical address**

   This ioctl is used to get the physical address of the allocated buffer.

   Ioctl: VIDIOC_QUERYBUF.

   It takes a pointer to the instance of v4l2_buffer structure as an argument. You must specify buffer type (V4L2_BUF_TYPE_VIDEO_OUTPUT), buffer index, and memory type (V4L2_MEMORY_MMAP) at the time of querying.

   Example:

```
/* allocate buffer by VIDIOC_REQBUFS */

/* structure to query the physical address of allocated buffer */
struct v4l2_buffer buffer;

buffer.index  = 0; /* buffer index for querying -0 */
buffer. type  = V4L2_BUF_TYPE_VIDEO_OUTPUT;
buffer.memory = V4L2_MEMORY_MMAP;

if (ioctl(fd, VIDIOC_QUERYBUF, &buffer) == -1) {
        printf("buffer query error.\n");
        close(fd);
        exit(-1);
}
/* The buffer.m.offset contains the physical address returned from driver */
```

3. **Mapping kernel space address to user space**

   Mapping the kernel buffer to the user space can be done via mmap. You can pass buffer size and physical address of buffer for getting the user space address.

```
/* allocate buffer by VIDIOC_REQBUFS */

/* query the buffer using VIDIOC_QUERYBUF */

/* addr hold the user space address */
Int addr;
Addr = mmap(NULL, buffer.size,PROT_READ | PROT_WRITE, MAP_SHARED,
         fd, buffer.m.offset);
/* buffer.m.offset is same as returned from VIDIOC_QUERYBUF */
```

The following videobuf_queue_ops are implemented for buffer management:

- **buf_setup** - initialized to davinci_buffer_setup
- **buf_prepare** - initialized to davinci_buffer_prepare
- **buf_queue** - initialized to davinci_buffer_queue
- **buf_release** - initialized to davinci_buffer_release
- **buf_config** - initialized to davinci_buffer_config

The driver also uses APIs provided by the videobuf driver for implementing buffer management.

2.3.3 IOCTL Handling

2.3.3.1 Query Capabilities

This ioctl is used to identify kernel devices compatible with V4L2 specification and to obtain information about individual hardware capabilities.

Ioctl: VIDIOC_QUERYCAP.

Capabilities can be video display (V4L2_CAP_VIDEO_OUTPUT) and streaming (V4L2_CAP_STREAMING).

It takes a pointer to the v4l2_capability structure as an argument. Capabilities can be accessed by the capabilities field of the v4l2_capability structure.

Example:

```
struct v4l2_capability capability;
ret = ioctl(fd, VIDIOC_QUERYCAP, &capability);
if(ret) {
printf("Cannot do QUERYCAP\n");
    return -1;
}
if(capability.capabilities & V4L2_CAP_VIDEO_OUTPUT) {
printf("Display capability is supported\n");
}
if(capability.capabilities & V4L2_CAP_STREAMING) {
    printf("Streaming is supported\n");
}
```

2.3.3.2 Output and Standard handling

The DaVinci Video Driver defines SysFs attributes for handling output and standard switching. For details, see the *LSP 1.20 DaVinci Video Sysfs User's Guide* (SPRUEL6). This guide lists all available outputs and standards defined by the driver. For this reason, the following IOCTLs are removed from V4L2 Display Driver:

```
VIDIOC_ENUMOUTPUT
VIDIOC_S_OUTPUT
VIDIOC_G_OUTPUT
VIDIOC_ENUMSTD
VIDIOC_QUERYSTD
VIDIOC_S_STD
VIDIOC_G_STD
```

2.3.3.3 Format Enumeration

This ioctl is used to enumerate the information of format (horizontal pitch/pixel length, buffer size, pixel format, etc.) that are supported by the current encoder.

Ioctl: VIDIOC_ENUM_FMT.

As per the current encoder's interface type, it fills the structure. The image format types currently supported are: V4L2_PIX_FMT_UYVY, the former being the default. It takes the pointer to the instance of the v4l2_fmtdesc structure as an output parameter.

Example:

```
struct v4l2_fmtdesc fmt;
I = 0;
while(1) {
    fmt.index = I;
    ret = ioctl(fd, VIDIOC_ENUM_FMT,& fmt);
    if(ret) {
        break;
    }
    printf("description = %s\n",fmt.description);
    if(fmt.type == V4L2_BUF_TYPE_VIDEO_OUTPUT)
        printf("Video Output type\n");
    if(fmt.pixelformat == V4L2_PIX_FMT_UYVY)
        printf("V4L2_PIX_FMT_UYVY\n");
    I++;
}
```

2.3.3.4 Set Format

This ioctl is used to set format for current encoder.

Ioctl: VIDIOC_S_FMT.

The driver validates the output parameters. The driver returns an error if parameters are not valid; otherwise, the driver configures these parameters. It takes the pointer to instance of the v4l2_format structure as an output parameter. Format type is set as V4L2_BUF_TYPE_VIDEO_OUTPUT for the display driver. The v4l2_format structure contains parameters like pixel format, height and width of the image, and field type. The driver calculates the bytes per line and size image based on the hardware capabilities and the application can retrieve the same using the Get Format ioctl. This is applicable when the driver-buffer mode is used as discussed in Section 2.3.2. If the user-buffer mode is used, the application needs to fill in sizeimage and bytesperline and the driver calculates the width and height of the image as follows.

```
width  = bytesperline/2
height = sizeimage/bytesperline
```

Example:

```
struct v4l2_format fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = height;
fmt.fmt.pix.width = width;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if(ret) {
    perror("VIDIOC_S_FMT\n");
    close(fd);
    return -1;
}
```

2.3.3.5 Get Format

This ioctl is used to get format for current encode

Ioctl: VIDIOC_G_FMT.

The driver provides format parameters in the structure pointer passed as an argument.

It takes the pointer to an instance of the v4l2_format structure as an output parameter. Format type is set as V4L2_BUF_TYPE_VIDEO_OUTPUT for the display driver. The v4l2_format structure contains parameters like pixel format, height and width of the image, size of image, bytes per line, and field type.

Example:

```
struct v4l2_format fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
ret = ioctl(fd, VIDIOC_G_FMT, &fmt);
if(ret) {
    perror("VIDIOC_G_FMT\n");
    close(fd);
    return -1;
}
```

2.3.3.6 Try Format

This ioctl is used to validate the format for current encoder.

Ioctl: VIDIOC_TRY_FMT.

The driver returns an error if parameters are not valid. It takes a pointer to an instance of the v4l2_format structure as an output parameter. Format type is set as V4L2_BUF_TYPE_VIDEO_OUTPUT for display driver. The v4l2_format structure contains parameters like pixel format, height and width of the image, and field type. The driver calculates the bytesperline and sizeimage based on the hardware capabilities and the application can retrieve the same using the Get Format ioctl.

Example:

```
struct v4l2_format fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = height;
fmt.fmt.pix.width = width;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_TRY_FMT, &fmt);
if(ret) {
    perror("VIDIOC_TRY_FMT \n");
    close(fd);
    return -1;
}
```

2.3.3.7 Query Crop Capabilities

This ioctl is used to query the crop capabilities of the driver. Applications use this to query the cropping limits, the pixel aspect of the image, and to calculate scale factor.

```
struct v4l2_cropcap cropcap;

memset(&cropcap,0,sizeof (cropcap));

ret = ioctl(fd, VIDIOC_CROPCAP, &cropcap);
if(ret) {
    perror("VIDIOC_CROPCAP\n");
    close(fd);
    return -1;
}
```

2.3.3.8 Set Crop

This ioctl is used to set bounds for the crop rectangle that is the target rectangle. This enables the application to display part of the image on the display or display it zoomed or expanded within the display bounds

The driver validates the crop parameters to be within the display bounds. The driver returns an error if parameters are not valid otherwise the driver configures these parameters with zoom/expansion, as desired.

It takes a pointer to an instance of the v4l2_crop structure as an output parameter. Crop type is set as V4L2_BUF_TYPE_VIDEO_OUTPUT for the display driver. The v4l2_crop structure contains parameters like top, left, and height and width of the crop area to be displayed.

The S_CROP parameters, height and width, and S_FMT image width and height are used by the driver to calculate the zoom factor and expansion factor required. To use this application, first set the image format by calling S_FMT and then call S_CROP to display it cropped or zoomed/expanded.

- The following zoom factors are used: 2x and 4x zoom for horizontal and vertical direction for all standards.
- For NTSC display: the display can be expanded by 9/8 in the horizontal direction to display the image in square pixels.
- For PAL display: the display can be expanded horizontally by 9/8 and vertically by 6/5 to display the image in square pixels.

Scenario 1: **Crop -** Display a portion of the image at given coordinates.

In this scenario, S_CROP is called to display part of the image.

Example:

```
struct v4l2_format fmt;
struct v4l2_crop crop;

fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = image_height;
fmt.fmt.pix.width = image_width;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if(ret) {
    perror("VIDIOC_S_FMT\n");
    close(fd);
    return -1;
}

// display half of 1/4th of the image area starting at 0,0
crop.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
crop.c.height = image_height/2;
crop.c.width = image_width/2;
crop.c.top = 0;
crop.c.left = 0;

ret = ioctl(fd, VIDIOC_S_CROP, &crop);
if(ret) {
    perror("VIDIOC_S_CROP\n");
    close(fd);
    return -1;
}
```

Scenario 2: **Zoom -** Display the image zoomed.

The application can zoom an image displayed by a factor of 2x or 4x. To do this, S_CROP height and width are set to 2x the values used in S_FMT. This results in the image zoomed and displayed at twice the original size. A similar action takes place if S_CROP values are provided for 4x zoom.

Example:

```
struct v4l2_format fmt;
struct v4l2_crop crop;
```

```
int image_height, image_width, zoom_factor;


image_height = 240;
image_width = 352;

fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = image_width;
fmt.fmt.pix.width = image_height;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if(ret) {
    perror("VIDIOC_S_FMT\n");
    close(fd);
    return -1;
}

// display image zoomed by 2x in both horizontal and vertical direction positioned at 0,0 on the
display
zoom_factor = 2;
crop.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
crop.c.height = image_height*zoom_factor;
crop.c.width = image_width*zoom_factor;
crop.c.top = 0;
crop.c.left = 0;

ret = ioctl(fd, VIDIOC_S_CROP, &crop);
if(ret) {
    perror("VIDIOC_S_CROP\n");
    close(fd);
    return -1;
}
```

Scenario 3: **Expansion** - Display the image expanded.

To display a VGA image as square pixels on a NTSC display, set S_FMT width and height to VGA bounds and width and height in S_CROP to NTSC bounds. This will result in a square-pixel display of VGA image. A similar action takes place for PAL, where expansion is done for both horizontal and vertical direction based on S_CROP values.

Example:

```
struct v4l2_format fmt;
struct v4l2_crop crop;
int image_height, image_width;


image_height = 480;
image_width = 640;

fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = image_width;
fmt.fmt.pix.width = image_height;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if(ret) {
    perror("VIDIOC_S_FMT\n");
    close(fd);
    return -1;
}

// display image expanded to get square pixel on a NTSC display

crop.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
crop.c.height = image_height;
crop.c.width = image_width*9/8;

crop.c.top = 0;
crop.c.left = 0;

ret = ioctl(fd, VIDIOC_S_CROP, &crop);
```

```
if(ret) {
    perror("VIDIOC_S_CROP\n");
    close(fd);
    return -1;
}
```

2.3.3.9 Get Crop

This ioctl is used to get crop rectangle bounds.

Ioctl: VIDIOC_G_CROP.

The driver provides crop bounds currently used in the driver.

It takes a pointer to an instance of the v4l2_crop structure as an output parameter. Crop type is set as V4L2_BUF_TYPE_VIDEO_OUTPUT for the display driver. The v4l2_crop structure contains parameters like top, left, and height and width of the crop area displayed.

Example:

```
struct v4l2_crop crop;
crop.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
ret = ioctl(fd, VIDIOC_G_CROP, &crop);
if(ret) {
    perror("VIDIOC_S_CROP\n");
    close(fd);
    return -1;
}
printf ("top = %d\n",crop.c.top);
printf ("left = %d\n",crop.c.left);
printf ("height = %d\n",crop.c.height);
printf ("width = %d\n",crop.c.width);
```

2.3.3.10 Queue Buffer

This ioctl is used to place the buffer in the buffer queue.

Ioctl: VIDIOC_QBUF.

The application has to specify the buffer type (V4L2_BUF_TYPE_VIDEO_OUTPUT), buffer index, and memory type (V4L2_MEMORY_MMAP or V4L2_MEMORY_USERPTR) at the time of queuing.

If this ioctl is called with the file descriptor, with which VIDIOC_REQBUF has not been done, the driver returns an error.

The driver places the buffer in the queue, if the buffer queue is not empty.

If queue is empty, then the driver directly writes the buffer address to the hardware registers. It takes a pointer to an instance of the v4l2_ buffer structure as an output parameter.

Example:

```
struct v4l2_buffer buf;
buf.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
buf.type = V4L2_MEMORY_MMAP;
buf.index = 0;
ret = ioctl(fd, VIDIOC_QBUF, &buf);
if(ret) {
    perror("VIDIOC_QBUF\n");
    close(fd);
    return -1;
}
```

2.3.3.11 Dequeue Buffer

This ioctl is used to dequeue the buffer in the buffer queue.

Ioctl: VIDIOC_DQBUF.

The application has to specify the buffer type (V4L2_BUF_TYPE_VIDEO_OUTPUT) and memory type (V4L2_MEMORY_MMAP or V4L2_MEMORY_USERPTR) at the time of dequeueing.

If this ioctl is called with the file descriptor, with which VIDIOC_REQBUF has not been done, the driver returns an error.

The driver places the buffer in the queue, if the buffer queue is not empty.

If the channel is opened in non-blocking mode, this ioctl call returns immediately if there is no buffer in the queue.

It takes a pointer to an instance of the v4l2_ buffer structure as an output parameter.

Example:

```
struct v4l2_buffer buf;
buf.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
buf.type = V4L2_MEMORY_MMAP;
ret = ioctl(fd, VIDIOC_DQBUF, &buf);
if(ret) {
    perror("VIDIOC_DQBUF\n");
    close(fd);
    return -1;
}
```

2.3.3.12 Stream On

This ioctl is used to start video display functionality.

Ioctl: VIDIOC_STREAMON.

If this ioctl is called with the file descriptor, with which VIDIOC_REQBUF has not been done, the driver returns an error.

If streaming is already started, this ioctl call returns an error.

Example:

```
ret = ioctl(fd, VIDIOC_STREAMON, NULL);
if(ret) {
    perror("VIDIOC_STREAMON \n");
    close(fd);
    return -1;
}
```

2.3.3.13 Stream Off

This ioctl is used to stop video display functionality.

Ioctl: VIDIOC_STREAMOFF.

If this ioctl is called with the file descriptor, with which VIDIOC_REQBUF has not been done, the driver returns an error.

If streaming is yet not started, this ioctl call returns an error.

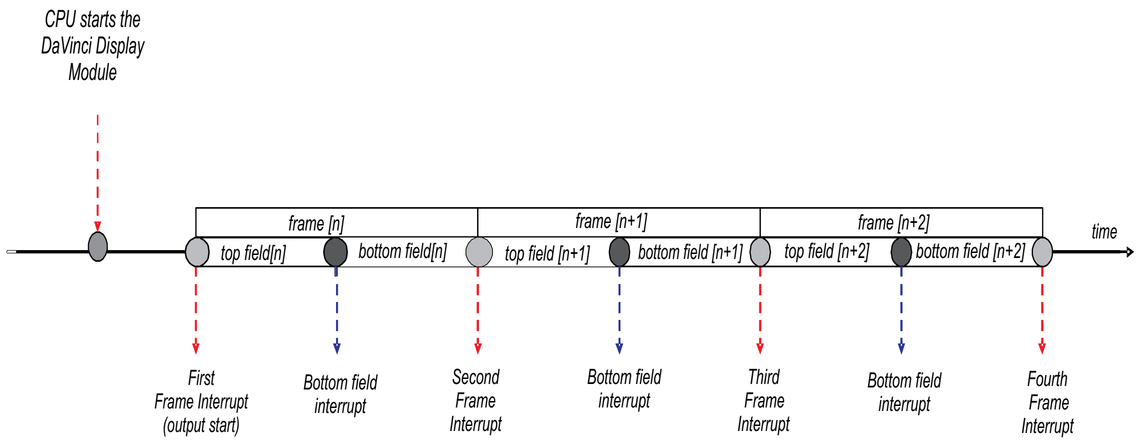
Example:

```
ret = ioctl(fd, VIDIOC_STREAMOFF, NULL);
if(ret) {
    perror("VIDIOC_STREAMOFF \n");
    close(fd);
    return -1;
}
```

2.4 Interrupt Handling

- As part of the initialization, the driver configures the software channel for device 2 and 3 to give an interrupt after single field/frame display. The OSD display manager registers the IRQ8 (VENCINT) handler for field or frame interrupt handling. The V4L2 driver registers the interrupt callback function (davinci_display_isr) with the display manager using the API, davinci_disp_register_callback(), for these events: DAVINCI_DISP_FIRST_FIELD (top field), DAVINCI_DISP_SECOND_FIELD (bottom field), and DAVINCI_DISP_END_OF_FRAME. The first two events are for handling interlaced display and the last one, for progressive display. In the call back routine, the driver changes the data buffer pointer in the VPBE controller register according to frame format (field format/frame format/progressive scan) and changes the current frame status.
- The address of the first frame is configured when the application calls VIDIOC_STREAMON IOCTL.
- As shown in Figure 5, the first interrupt comes when the DaVinci Display device starts storing a frame in the SDRAM.
- In the interrupt handler, the driver configures an address for the next frame in the VPBE registers, if frame format is progressive.
- If frame format is interlaced in the interrupt handler, the driver configures an address for the next frame in VPBE registers when an interrupt for the bottom field is received. For the top field interrupt, it marks that the display is completed for the current frame.
- After a single frame display interrupt, the driver gets system time using the jiffies timer. This time displays time for a particular frame. Applications can get this empty buffer with display time by calling IOCTL with the VIDIOC_DQBUF command.
- If there is only a single buffer in the incoming queue of the driver, the driver does not update the VPBE registers and the DaVinci Display device displays the same frame until the application calls VIDIOC_QBUF IOCTL. In the VIDIOC_QBUF IOCTL, the driver sets the address of buffer to be queued in the VPBE registers if the buffer queue is empty. So, when the next interrupt comes, a buffer is already available to the device for display. In this way, the driver can save itself from displaying the same frame once. There is a restriction here. If the application does not call VIDIOC_QBUF ioctl after the buffer queue becomes empty and before the next interrupt, the same frame is displayed.

Figure 5. Relationship Between First Interrupt and Incoming Data



3 User Interface

3.1 Data-Structures

3.1.1 Request Buffer Structure for V4L2

```
struct v4l2_requestbuffers
{
    __u32                   count;
    enum v4l2_buf_type      type;
    enum v4l2_memory        memory;
    __u32                   reserved[2];
};
```

Relevant Fields for DaVinci Display Driver

count	Number of buffers requested.
type	Here, it will be always V4L2_BUF_TYPE_VIDEO_OUTPUT.
memory	V4L2_MEMORY_MMAP or V4L2_MEMORY_USERPTR

3.1.2 Buffer Structure for V4L2

```
struct v4l2_buffer
{
    __u32                  index;
    enum v4l2_buf_type     type;
    __u32                  bytesused;
    __u32                  flags;
    enum v4l2_field        field;
    struct timeval         timestamp;
    struct v4l2_timecode   timecode;
    __u32                  sequence;

    /* memory location */
    enum v4l2_memory       memory;
    union {
        __u32              offset;
        unsigned long      userptr;
    } m;
    __u32                  length;
    __u32                  output;
    __u32                  reserved;
};
```

Relevant Fields for DaVinci Display Driver

index	Index of the buffer.
type	Type of the data stream; here, it will be always V4L2_BUF_TYPE_VIDEO_OUTPUT.
memory	V4L2_MEMORY_MMAP or V4L2_MEMORY_USERPTR
bytesused	The number of bytes occupied by the data in the buffer.
field	Indicates the field order of the image in the buffer.
m.userptr	When memory is V4L2_MEMORY_USERPTR, this is a pointer to the buffer in virtual memory set by the application.
m.offset	When memory is V4L2_MEMORY_MMAP, this is the offset of the buffer from the start of the device memory.
length	Size of the buffer.

3.1.3 Query Capability Structure for V4L2

```
struct v4l2_capability
{
    __u8    driver[16];
    __u8    card[32];
    __u8    bus_info[32];
    __u32   version;
    __u32   capabilities;
    __u32   reserved[4];
};
```

Relevant Fields for DaVinci Display Driver

driver	*DaVinci Display*
card	*DM355 EVM* or *DM644X EVM*
bus_info	*Platform*
version	Version code for DaVinci Display module.
capabilities	V4L2_CAP_VIDEO_OUTPUT | V4L2_CAP_STREAMING

3.1.4 Format Description Structure for V4L2

```
struct v4l2_fmtdesc
{
    __u32               index;             /* Format number      */
    enum v4l2_buf_type  type;              /* buffer type        */
    __u32               flags;
    __u8                description[32];   /* Description string */
    __u32               pixelformat;       /* Format fourcc      */
    __u32               reserved[4];
};
```

Relevant Fields for DaVinci Display Driver

index	Index of the supported formats
type	Type of the data stream; here, it will be always V4L2_BUF_TYPE_VIDEO_OUTPUT.
description	Description of the format, a NULL-terminated ASCII string. This information is intended for the user.
pixelformat	The image format identifier: V4L2_PIX_FMT_UYVY

3.1.5 Format Structure for V4L2

```
struct v4l2_format
{
    enum v4l2_buf_type type;
    union
    {
        struct v4l2_pix_format    pix;
        struct v4l2_window        win;
        struct v4l2_vbi_format    vbi;
    } fmt;
};
struct v4l2_pix_format
{
    __u32                 width;
    __u32                 height;
    __u32                 pixelformat;
    enum v4l2_field       field;
    __u32                 bytesperline;
    __u32                 sizeimage;
    enum v4l2_colorspace  colorspace;
    __u32                 priv;
};
```

Relevant Fields for DaVinci Display Driver

type	Type of the data stream; here, it will be always V4L2_BUF_TYPE_VIDEO_OUTPUT.
fmt.pix.pixelformat	The pixel format or type of compression set by the application: V4L2_PIX_FMT_UYVY.
fmt.pix.field	Indicates the field order of the image in the buffer.
width	Width of the image in pixels. Filled by the application or the driver. The application fills this when the driver-provided buffer is used and the driver fills it when the user-provided buffer is used; should be a multiple of 16.
height	Height of the image in pixels. Filled by the application or the driver. The application fills this when the driver-provided buffer is used and the driver fills it when the user-provided buffer is used; should be a multiple of 2, for interlaced display.
fmt.pix.bytesperline	Distance, in bytes, between the left-most pixels in two adjacent lines. Set by the driver or the application. The driver fills it for the driver-provided buffer and the application fills it for the user-provided buffer.
fmt.pix.sizeimage	Size, in bytes, of the buffer to hold a complete image. Set by the driver or the application. The driver fills it for the driver-provided buffer and the application fills it for the user-provided buffer.

3.1.6 Crop Structure for V4L2

```
struct v4l2_rect
{
    __u32         left;
    __u32         top;
    __u32         width;
    __u32         height;
};

struct v4l2_crop
{
    enum v4l2_buf_type type;
    struct v4l2_rect   c;
};
```

Relevant Fields for DaVinci Display Driver

type	Type of the data stream; here, it will be always V4L2_BUF_TYPE_VIDEO_OUTPUT.
left	Horizontal offset of the top, left corner of the rectangle, in pixels. Set by the application or the driver.
top	Vertical offset of the top, left corner of the rectangle, in pixels. Set by the application or the driver.
width	Width of the rectangle, in pixels. Set by the application or the driver.
height	Height of the rectangle, in pixels. Set by the application or the driver.

3.2 Enumerations and Defines

3.2.1 Pixel Formats

This describes the pixel format supported by DaVinci Display Driver.

```
#define V4L2_PIX_FMT_UYVY
```

This pixel format is for UYVY.

3.2.2 Crop Bounds and Pixel Aspects

These are the values returned for CROPCAP.

Aspects

```
#define DAVINCI_DISPLAY_PIXELASPECT_NTSC         {11, 10}
#define DAVINCI_DISPLAY_PIXELASPECT_PAL          {54, 59}
#define DAVINCI_DISPLAY_PIXELASPECT_SP           {1, 1}
```

SP pixel aspects are used for HD displays.

Bounds

```
#define DAVINCI_DISPLAY_WIN_NTSC     {0, 0, 720, 480}
#define DAVINCI_DISPLAY_WIN_PAL      {0, 0, 720, 576}
#define DAVINCI_DISPLAY_WIN_640_480  {0, 0, 640, 480}
#define DAVINCI_DISPLAY_WIN_640_400  {0, 0, 640, 400}
#define DAVINCI_DISPLAY_WIN_640_350  {0, 0, 640, 350}
#define DAVINCI_DISPLAY_WIN_720P     {0, 0, 1280, 720}
#define DAVINCI_DISPLAY_WIN_720P     {0, 0, 1920, 1080}
```

3.3 API Specification

open

Description: Opens the device driver for processing. The driver supports single I/O instance and multiple control instances of each of the channels.

Initializes the DAVINCI channel.

Reads the current display mode from the encoder manager and sets the default image and crop parameters.

Prototype

```
int fd = open(Devicename,mode);
```

Arguments

Devicename [IN]	It can be /dev/video2 or /dev/video3
mode [IN]	O_RDWR [| O_NONBLOCK] Here, O_NONBLOCK is optional. If it is non-blocking at the time of the buffer dequeue, it returns immediately, even if it does not get a buffer.

Return Value Zero on success or negative, if an error occurred.

-ENOMEM error, if no memory is available to create an instance of the channel.

close

Description: Closes the open channel handle. While closing, if an I/O instance tries to close the handle while other descriptors are still open for same channel, an error is returned.

Prototype

```
close(fd);
```

Arguments

fd [IN]	File descriptor returned from open.

Return Value Zero on success.

-EAGAIN, if an I/O instance tries to close the handle while other descriptors are still open for same channel.

mmap

Description: Maps the kernel space buffer to user space.

Prototype

```
void * mmap(void *,size_t image_size,int prot,int flags,int fd,off_t offset)
```

Arguments

*void**	Generally, NULL.
image_size [IN]	Buffer size that needs to be mapped.
flags [IN]	PROT_SHARED
fd [IN]	File descriptor.
offset [IN]	Physical address of buffer.

Return Value Zero on success or -EAGAIN on address not found.

munmap

Description: Unmaps the frame buffers that were previously mapped to user space, using mmap().

Prototype

```
void *(void *start,size_t length)
```

Arguments

image_size [IN]	Buffer size that needs to be mapped.

Return Value Zero on success or negative, if an error occurred.

VIDIOC_REQBUFS

Description: Used to allocate memory.

In order to convert a control channel instance into an I/O instance, the application needs to call this ioctl.

Prototype

```
int ret = ioctl(fd , VIDIOC_REQBUFS, &reqbuf);
```

Arguments

reqbuf [IN-OUT]	Pointer to the v4l2_requestbuffers structure.

User specifies the buffer type, number of buffers, and memory type as an output field.

Return Value Zero on success or negative, if an error occurred.

-EBUSY error, if an I/O instance already exists.

-EINVAL error, if the driver buffer is requested and zero buffers are allocated at the time of driver insertion.

VIDIOC_QUERYBUF

Description: Used to get the physical address of the allocated buffer.

Prototype `int ret = ioctl(fd, VIDIOC_QUERYBUF, &buffer);`

Arguments

buffer [IN-OUT]	Pointer to the v4l2_buffer structure.

User specifies the buffer type, buffer index, and memory type as an output field.

Return Value Zero on success or negative, if an error occurred.

VIDIOC_QUERYCAP

Description: Used to querying driver capabilities.

Prototype `int ret = ioctl(fd, VIDIOC_QUERYCAP, &capability);`

Arguments

capability [OUT]	Pointer to the v4l2_capability structure.

Return Value Zero on success or negative, if an error occurred.

VIDIOC_ENUM_FMT

Description: Used to enumerate the format information supported by the current display mode.

Prototype `int ret = ioctl(fd, VIDIOC_ENUM_FMT,& fmtdesc);`

Arguments

fmtdesc [IN-OUT]	Pointer to the v4l2_fmtdesc structure.

User specifies the index number.

Return Value Zero on success or negative, if an error occurred.

-EINVAL, if output index is not within range of all the supported standards.

VIDIOC_S_FMT

Description: Used to set the format for the current display mode.

Prototype `int ret = ioctl(fd, VIDIOC_S_FMT, &fmt);`

Arguments

fmt [IN]	Pointer to the v4l2_format structure.

Return Value Zero on success or negative, if an error occurred.

VIDIOC_G_FMT

Description: Used to get the format for the current display mode.

Prototype `int ret = ioctl(fd, VIDIOC_G_FMT, &fmt);`

Arguments

fmt [OUT]	Pointer to the v4l2_format structure.

Return Value Zero on success or negative, if an error occurred.

VIDIOC_TRY_FMT

Description: Used to validate the format for the current display mode.

Prototype `int ret = ioctl(fd, VIDIOC_TRY_FMT, &fmt);`

Arguments

fmt [IN]	Pointer to the v4l2_format structure.

Return Value Zero on success or negative, if an error occurred.

VIDIOC_CROPCAP

Description: Used to querying the crop capabilities of the driver.

Prototype `int ret = ioctl(fd, VIDIOC_CROPCAP, &capability);`

Arguments

capability [OUT]	Pointer to the v4l2_cropcap structure.

Return Value Zero on success or negative, if an error occurred.

VIDIOC_S_CROP

Description: Used to set the crop bounds for the target rectangle.

Prototype `int ret = ioctl(fd, VIDIOC_S_CROP, &crop);`

Arguments

crop [IN	Pointer to the v4l2_crop structure.

Return Value Zero on success or negative, if an error occurred.

VIDIOC_G_CROP

Description: Used to get the crop bounds of the target rectangle.

Prototype `int ret = ioctl(fd, VIDIOC_G_FMT, &crop);`

Arguments

buf [IN]	Pointer to the v4l2_buffer structure.

User specifies buffer type, buffer index, and memory type.

Return Value Zero on success or negative, if an error occurred.

-EACCES, if this is called through control fd and not with I/O fd.

VIDIOC_DQBUF

Description: Used to dequeue the buffer in the buffer queue.

Prototype `int ret = ioctl(fd, VIDIOC_DQBUF, &buf);`

Arguments

buf [OUT]	Pointer to the v4l2_buffer structure.

User specifies buffer type and memory type.

Return Value Zero on success or negative, if an error occurred.

-EACCES, if this is called through control fd and not with I/O fd

VIDIOC_STREAMON

Description: Used to start the video display functionality.

Prototype `int ret = ioctl(fd, VIDIOC_STREAMON, &buf);`

Arguments None

Return Value Zero on success or negative, if an error occurred.

-EACCES, if this is called through control fd and not with I/O fd.

-EBUSY, if streaming is already started.

-EIO, if buffer queue is empty.

VIDIOC_STREAMOFF

Description: Used to stop the video display functionality.

Prototype `int ret = ioctl(fd, VIDIOC_STREAMOFF, &buf);`

Arguments None

Return Value Zero on success or negative, if an error occurred.

-EACCES, if this is called through control fd and not with I/O fd.

-EBUSY, if streaming is not started.

4 Build and Installation

4.1 Build Steps

The V4L2 driver depends on the following modules for its low level functionality:

- DaVinci Encoder Manager and Encoder Modules
- DaVinci Display Manager

  The following section shows how to enables these modules at build time.

4.1.1 Building V4L2 and Other Dependent Drivers as Static Modules

The following steps describe the procedure to build DaVinci Display Driver support in the kernel as static modules:

1. Open menuconfig options from the kernel command prompt.
2. Select **Device Driver** → **Multimedia devices** → **Video For Linux**
3. Select the following driver modules as static <*> as shown below:
   - **DaVinci V4L2 Video Display** for the V4L2 driver.
   - **DaVinci Encoder Manager** support for the DaVinci Encoder Manager driver.
     – Choose 1 for **Max number of channels for Encoder Manager**
   - **DaVinci VPBE Encoder support** for display SDTV through the internal VPBE encoder driver-to-COMPOSITE/SVIDEO/COMPONENT outputs.
   - **Logic PD Encoder support** for display VGA through the logic PD LCD daughtercard-to-LCD output.
   - **THS8200 Encoder support** for display HDTV through the THS8200 daughtercard-to-COMPONENT1 output.
4. Save and exit to build the kernel.

```
Multimedia devices -

          <*> Video For Linux -

                    Video For Linux -
                         ----Video Adapers----
                         ………..
                         <*> Davinci V4L2 Video Display
                         <*> Davinci Encoder Manager support
                         (1) Max  number of channels for Encoder Manager
                         <*> Davinci VPBE Encoder support
                         <*> Logic PD Encoder support
                         <*> THS8200 Encoder support
                         ----Davinci Display manager
```

4.1.2 Building V4L2 and Other Dependent Drivers as Dynamically-Loadable Modules

The following steps describe the procedure to build DaVinci Display Driver support in the kernel as dynamically-loadable modules (see the note below for the restriction that applies).

1. Open menuconfig options from kernel command prompt.
2. Select **Device Driver** → **Multimedia devices** → **Video For Linux**
3. Select the following driver modules as static <M> as shown below:
   - **DaVinci V4L2 Video Display** for the V4L2 driver.
   - **Encoder Manager support** for the DaVinci Encoder Manager driver.
     – Choose 1 for **Max number of channels for Encoder Manager**
   - **DaVinci VPBE Encoder support** for display SDTV through the internal VPBE encoder driver-to-COMPOSITE/SVIDEO/COMPONENT outputs.
   - **Logic PD Encoder support** for display VGA through the logic PD LCD daughtercard-to-LCD output.

- **THS8200 Encoder support** for display HDTV through the THS8200 daughtercard-to-COMPONENT1 output.

4. Save and exit to build the kernel.

Multimedia devices -

```
          <*> Video For Linux -

                    Video For Linux -
                         ----Video Adapers----
                         ………..
                         <M> Davinci V4L2 Video Display
                         <M>  Davinci Encoder Manager support
                         (1) Max  number of channels for Encoder Manager
                         <M> Davinci VPBE Encoder support
                         <M> Logic PD Encoder support
                         <M> THS8200 Encoder support
                         ----Davinci Display manager
```

5. After building the kernel, do *make modules* to build the driver as dynamically-loadable modules.
6. The following .ko files are created:
   - davinci_display.ko
   - davinci_enc_mngr.ko
   - davinci_platform.ko
   - logicpd_encoder.ko
   - ths8200_encoder.ko
   - vpbe_encoder.ko
   - davinci_osd.ko

Note: The V4L2 driver shares the common modules (6.b to 6.g) with the FBDev driver. Since FBDev is suggested to be built as a static module, it is necessary to build these modules as static as well. So, if the V4L2 driver is used along with FBDev driver on the target system, build these modules as static modules. However, the V4L2 driver (6.a) can be built either as a static or a dynamically-loadable module.

4.2 Boot Arguments

The following boot arguments apply when the driver is built statically into the kernel:

```
davinci-display.video2_numbuffers=5
davinci-display.video3_numbuffers=5
davinci-display.video2_bufsize=691200
davinci-display.video3_bufsize=691200
```

where video2_numbuffers = 0 and video3_numbuffers = 0 to set the driver buffer count to zero (when the user pointer I/O method is used for streaming). A non-zero value allocates the specified number of buffers as follows:

```
1 <= videox_numbuffers <= 3    -> Three driver buffers will be allocated.
3 <  videox_numbuffers        -> Allocated driver buffer count will be videox_numbuffers, where x
is 2 or 3
```

4.3 Installation

The following examples show dynamic insertion an removal of the DaVinci Video Display Device Driver and all dependent modules. Perform insmod in the order shown:

```
>insmod davinci_osd.ko
>insmod davinci_platform.ko
>insmod davinci_enc_mngr.ko ch0_output=<output> ch0_mode=<mode>
>insmod vpbe_encoder.ko
>insmod logicpd_encoder.ko
>insmod ths8200_encoder.ko
>insmod davinci-display.ko  video2_numbuffers=3 video3_numbuffers=3  video2_bufsize=691200
```

```
video2_bufsize=691200
```

Perform the rmmod in the following order:

```
>rmmod davinci-display.ko
>rmmod ths8200_encoder.ko
>rmmod logicpd_encoder.ko
>rmmod vpbe_encoder.ko
>rmmod davinci_enc_mngr.ko
>rmmod davinci_platform.ko
>rmmod davinci_osd.ko
```

5 Example Applications

The following sample applications are provided to showcase the display functionality of the DaVinci Display as part of the release. These modes provide loopback of captured video to the display:

- NTSC
- PAL
- 480P-60
- 576P-50
- 640x480
- 720P-60
- 1080I-30

For usage details on the V4L2 loopback applications for SD and HD, please refer to

- `PSP_##_##_##_###/examples/v4l2/readme.txt`
- `PSP_##_##_##_###/examples/dm355/ipipe/readme.txt`