LSP 1.20 DaVinci Linux VPBE Frame Buffer Driver

User's Guide

Literature Number: SPRUEK9
March 2008

TEXAS INSTRUMENTS

Contents

List of Figures

List of Tables

LSP 1.20 DaVinci Linux VPBE Frame Buffer Driver

This guide introduces the DaVinci Linux VPBE Frame Buffer Device (FBDev) Driver by providing a brief overview of the driver and specifics concerning its use within a hardware/software environment. For LSP 1.20, the FBDev Driver is supported on the following EVMs: DM644x, DM355.

1 Features and System Requirements

This section describes the functional scope of the FBDev Driver and its feature set. The section also details the various deployment environments, hardware and software, that the FBDev Driver is presently supported on.

1.1 Supported Services and Features

The FBDev Driver provides the following functional services:

- Support for the window enable/disable option from the boot argument line.
- It is able to provide input to THS8200 daughtercard to output component HD signal (720p and 1080i).
- VPBE driver supports analog interfaces like svideo, component, and composite.
- Support for direct output to the LogicPD LCD panel.
- Support for runtime enable/disable of all video and OSD windows as well as the cursor window.
- Support for 24-bit graphics.
- Support for user-defined color look-up tables.
- Support for video, OSD, and cursor window parameter configuration.
- Support for attribute window.
- Support for 1-/2-/4-/8-bit bitmap windows.
- Support for non-standard window resolutions.

1.2 System Requirements

The FBDev Driver is supported on platforms characterized by the following software and hardware requirements.

The platform that supports the software requirements is Monta Vista Linux 2.6.10.

Hardware requirements are supported by:

- DM6446 and DM355 EVM Boards
- LogicPD LCD
- SD TV and HD TV
- Cables

2 FBDev Driver Design

2.1 Introduction

In LSP1.20, a new design of video driver architecture is introduced to make the coexistence of FBDev and V4L2 drivers possible. This chapter explains the rationale behind the new architecture and its main components. Details of each component are also discussed here.

2.2 New Video Driver Architecture

The FBDev and V4L2 drivers implement the lower layers of the corresponding framework. In the 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 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 the 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 be 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 models:
   - Changes confined to hardware layer.
   - Minimum changes to hardware-independent layers.
3. Re-use the encoder interface developed for the DM6467 EVM 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 1.

- Hardware-independent layer
- 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:

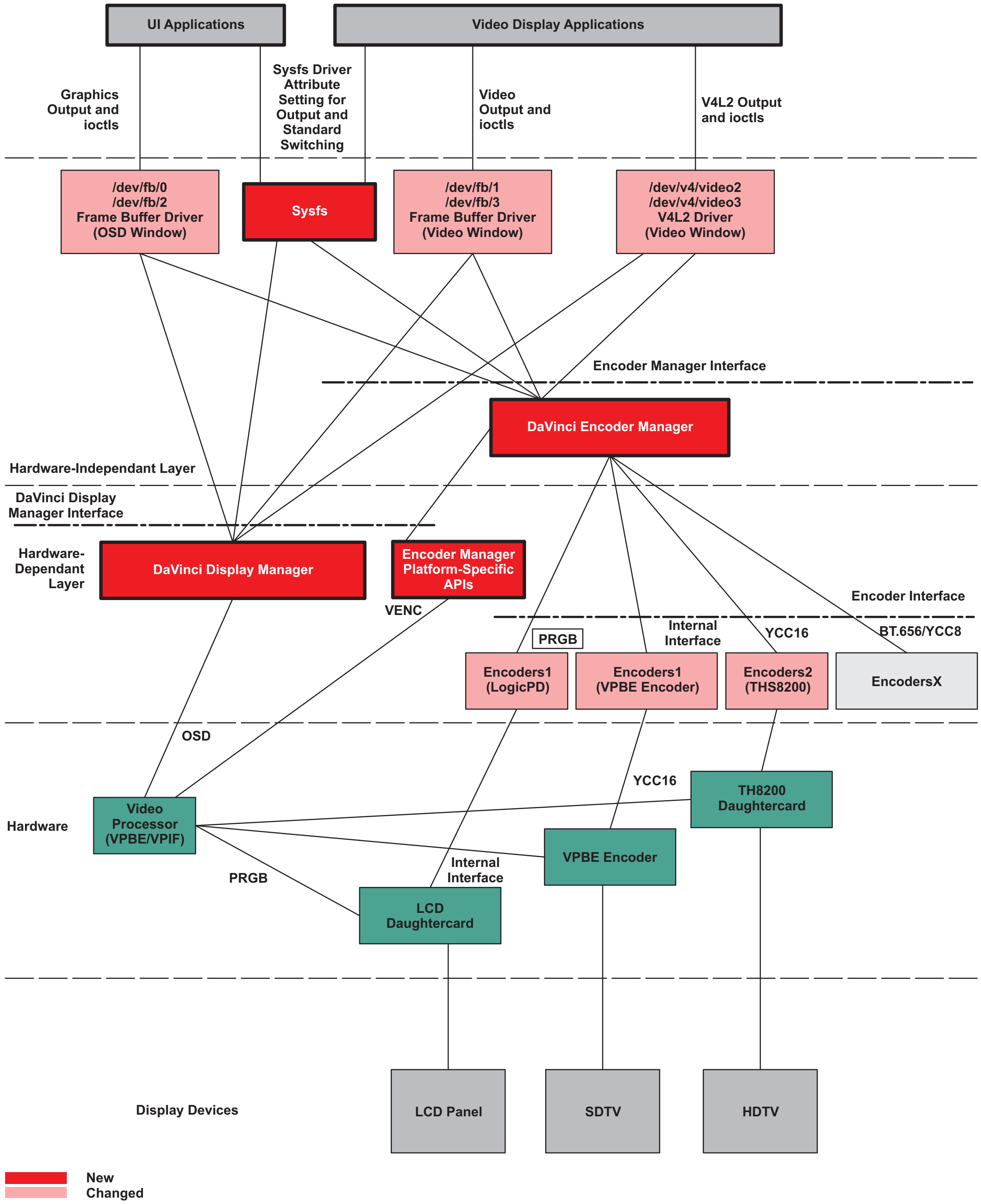
1. DaVinci display manager
2. Encoder manager platform APIs
3. Encoders

The following color coding is used in Figure 1:

- 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 replace them with standard APIs. Current V4L2 implementation used an encoder interface that was developed for the DM6467 EVM for easy integration of hardware encoders to the driver. However, it has been developed with a V4L2 bias. This implementation is re-used in this architecture by eliminating V4L2-specific definitions with standard C definitions. The following sections discuss the high-level details of the new modules introduced in this architecture.

Figure 1. DaVinci Video Display Driver Architecture



2.2.1 DaVinci Display Manager

The DaVinci Display Manager is responsible for the following functionality:

- Layer management. All OSD and video layers are initially owned by the display manager and 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 them at the 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.
- Service to the FBDev and V4L2 drivers for configuring the OSD hardware. This involves setting buffer address, line length, blending, zooming/scaling, window dimension, and other related functionality.
- Color look-up table management. This allows configuration of the RAM/ROM CLUTs for use and updates to the 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. Cursor position setting and blinking is also allowed.
- 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 section 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).

2.2.2 DaVinci Encoder Manager

The DaVinci Encoder Manager is responsible for the following:

- Managing the registration and de-registration of encoders that implement a set of API calls.
- Providing APIs 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 platforms need 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 1). The Encoder Manager hides the details of which encoder supports which standard and output and provides a set of generic APIs to invoke its services. Figure 2 shows how the encoders are interfaced to the SoC at the digital video port (shown as interface, which could be YCbCr/YCC8/YCC16/BT.656/PRGB/SRGB).

Figure 2. Encoder Interface

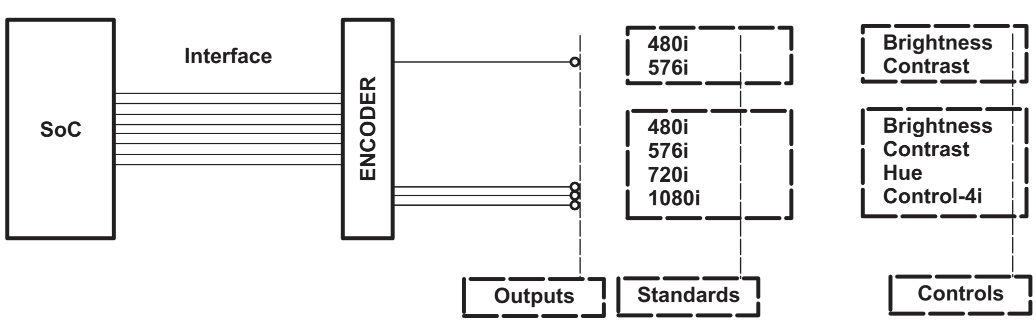


Table 1 lists the responsibilities of the Encoder Manager and encoders for each of the commands it services.

Table 1. Encoder Manager and Encoders for Each Command

Command	Encoder Manager	Encoder
Set output	Set current encoder to the 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	Enumerates 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.

2.2.3 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 output switching. Instead of abusing the FBDev interface with proprietary IOCTLs, it was decided to remove this functionality from V4L2 and FBDev and implement the same functionality 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.

2.3 FBDev Driver Design

Figure 3 shows the basic architecture of the FBDev Driver. The FBDev Driver invokes the services of the Display Manager and Encoder Manager modules which implement the low-layer functionality; e.g., interfacing with hardware. It also provides an FBDev front end to support setting up OSD and video windows and all the application interfaces that are visible to a user application; e.g., IOCTLs.

Figure 3. FBDEV Driver Architecture

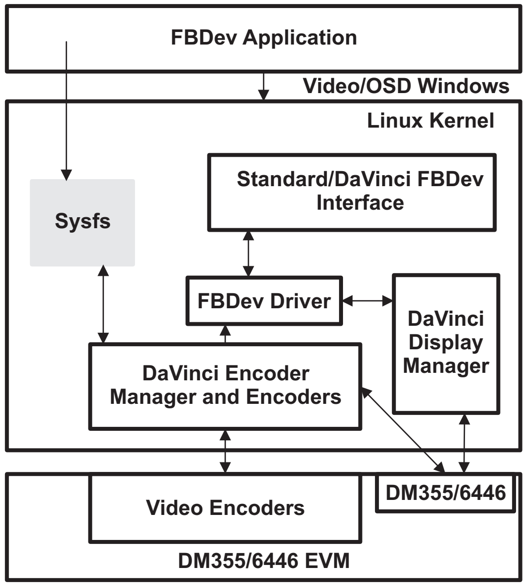
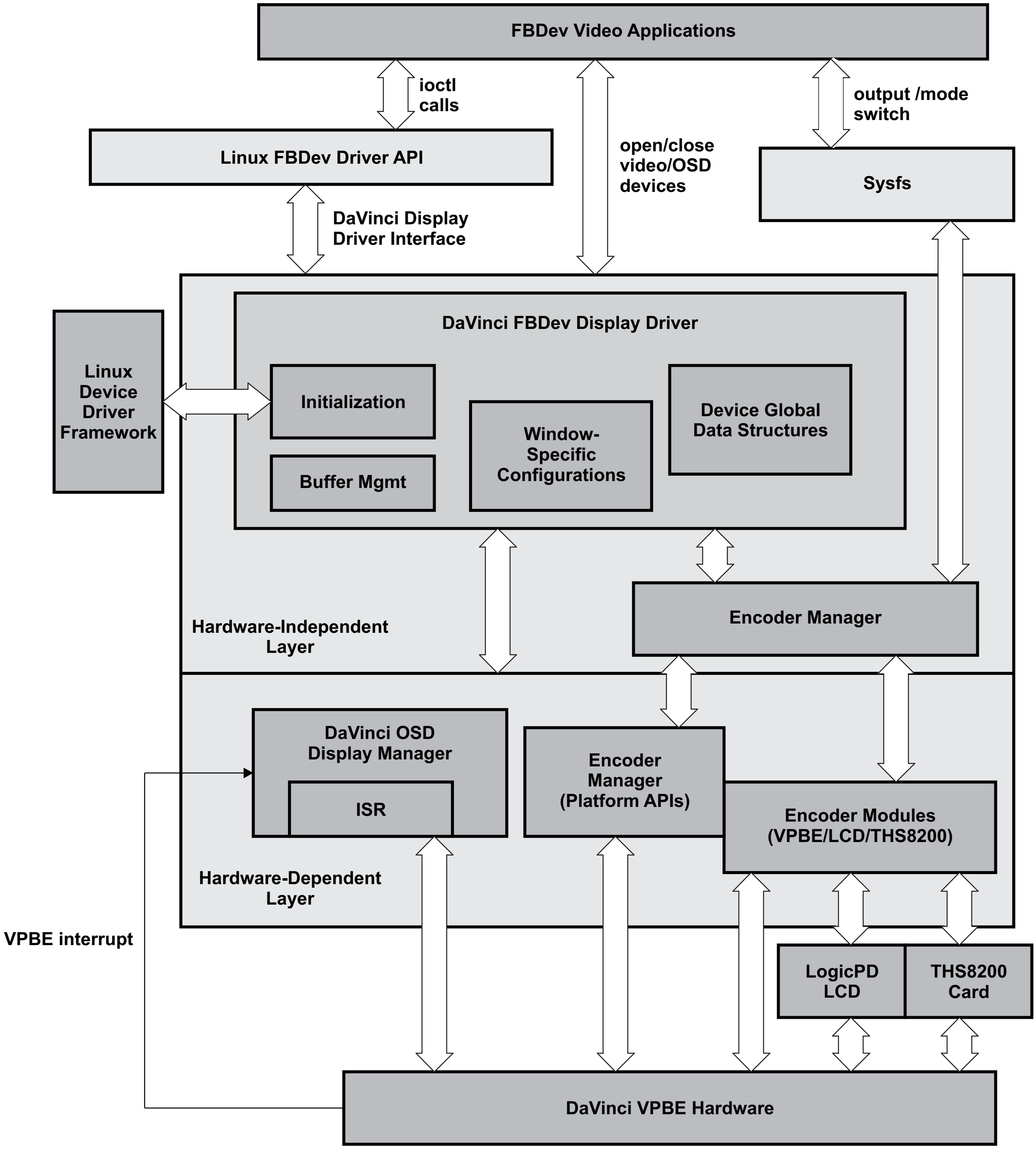


Figure 4 shows a more detailed view of how the FBDev driver components and their interactions with each other. It can be broken down into a hardware-independent layer and a hardware-dependent layer. The API services provided by the FBDev Driver are through:

- Standard file operation system calls (open/close/mmap).
- FBDev IOCTLs (standard and DaVinci-specific).
- FBDev Sysfs attributes (display output/mode switching).

Figure 4. DaVinci FBDev Driver



2.3.1 Hardware-Independent Layer

2.3.1.1 Display Driver

The Display Driver implements various Linux OS routines that provide system initialization and device configuration as well as services to the applications (via ioctl calls). It also invokes services from the hardware-dependent layer below it to perform the actual task in the hardware. On system startup, the driver registers itself with the kernel device driver database and initializes devices with either the system-default or the kernel boot argument supplied parameters. On system shutdown, the driver un-registers itself from the database and performs a clean-up task. The supported ioctls are both the standard open source and DaVinci-proprietary. The Device Driver also maintains the device global data structures that are used in the initialization and configuration for display windows. It opens and closes the device once an application is accessing via *open* and *close*. Frame buffer management also manages memory mapping (*mmap*) and un-mapping (*munmap*) between the applications and the kernel.

2.3.1.2 Encoder Manager

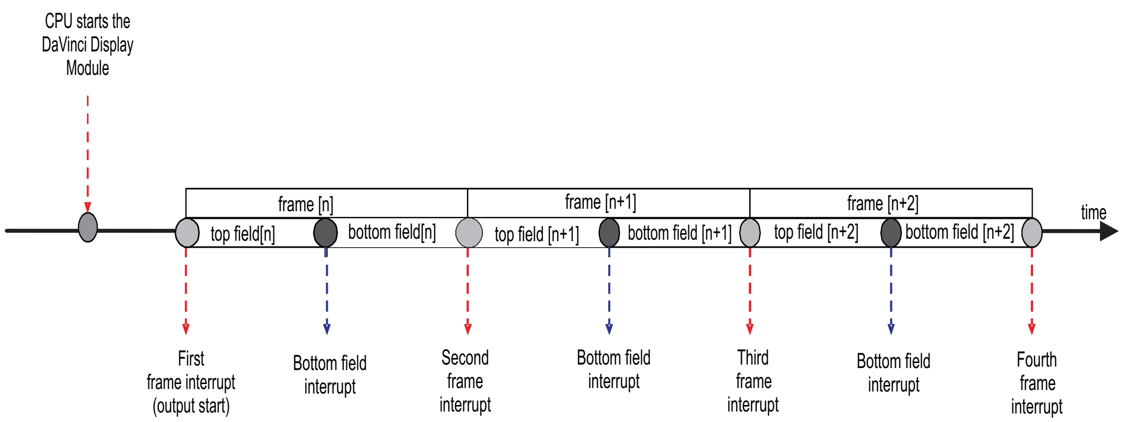
The platform-independent part of the Encoder Manager is discussed in Section 2.2.2.

2.3.2 Hardware-Dependent Layer

2.3.2.1 Interrupt Service Routine (ISR)

In the hardware-dependent layer of the FBDev Display Driver, the OSD Display Manager has an ISR for the DaVinci Display Driver. The ISR registers the IRQ8 (VENCINT) for the field or frame interrupt handling. The value of the FIDST bit field in the VPBE VSTAT register indicates the top or bottom field for interlaced display (see Figure 5). The driver software needs to update the address and line offset registers only during the bottom field for proper display of interlaced frames. The end-of-frame event is generated when FIDST=1. The DaVinci FBDev driver registers a call-back function for the End of Frame event at system initialization, so when each time when this event is received, the call-back function is executed. The interface ioctl to this callback function is FBIO_WAITFORVSYNC. The application takes the responsibility of managing the frame buffers and copying user data into the frame buffer. To display a frame, it calls FBIO_WAITFORVSYNC ioctl, which is blocked while waiting for the interrupt to come. When the callback is executed, it causes the application to unblock, and it can immediately call FBIOPAN_DISPLAY ioctls to write frame buffer address to the driver. The driver then writes the frame buffer address into VPBE registers for hardware to display the frame during bottom field. The address gets latched at the beginning of the frame that follows. The following figure shows the relationship between first interrupt and the incoming data.

Figure 5. Relationships Between First Interrupt and Incoming Data



2.3.2.2 Other Components

For the discussions of other components in this layer, like Encoder Manager platform APIs and Encoder modules, see Section 2.2.

3 FBDev Build and Configuration

This section discusses the FBDev Driver build and which software and hardware components are used to complete a successful installation. Also, it discusses how the FBDev driver is configured at boot time.

3.1 File Components

This section details the files and directory structure of the final installed DaVinci VPBE FBDev Driver in the system.

The open-source (standard) FBDev driver file is in the `/opt/montavista/pro/devkit/lsp/ti-davinci/drivers/video` directory. The `fbmem.c` file mainly implements the routines for generic ioctls FBDev API interface.

The DaVinci FBDev driver file is in the `/opt/montavista/pro/devkit/lsp/ti-davinci/drivers/video/davinci` directory. The `davincifb.c` file implements the hardware-independent FBDev driver, including but not limited to, driver registration, initialization, display window configuration, and their reverse operations.

The generic FBDev header file is in the `/opt/montavista/pro/devkit/lsp/ti-davinci/include/linux` directory to be included for building the FBDev display driver. The `fb.h` file also lists the generic (standard) ioctls to be included for building user applications.

The header file is in the `/opt/montavista/pro/devkit/lsp/ti-davinci/include/video` directory for building user applications. The `davincifb_ioctl.h` file lists the DaVinci-specific ioctls.

The following DaVinci-specific driver header files are in the `/opt/montavista/pro/devkit/lsp/ti-davinci/include/video` directory for building the display driver:

- `davincifb.h`
- `davinci_osd.h`
- `davinci_vpbe.h`

The following files are in the`/opt/montavista/pro/devkit/lsp/ti-davinci/drivers/media/video/davinci` directory. They are all part of the display driver:

- `davinci_enc_mngr.c` - DaVinci Encoder Manager, platform independent, also Sysfs attributes
- `davinci_platform.c` - DaVinci Encoder Manager, platform dependent
- `davinci_osd.c` - DaVinci (OSD) Display Manager
- `vpbe_encoder.c` - VENC Encoder Module
- `logicpd_encoder` - LogicPD Encoder Module
- `ths8200_encoder.c` - THS8200 (HD) Encoder Module

3.2 Development Tools Environment

This section describes the development tools environment(s) for software development. It describes the tools used for each supported environment.

3.2.1 Development Tools

Install the following tools, in the order given, to set up the development environment:

- Development tool/component, MVL 401 version 2.6.10
- MontaVista Linux Toolchain - arm_v5t_le-

3.3 Build

3.3.1 Build Steps

Follow the steps below to enable FBDev support in the system:

Step 1. Choose your default kernel configuration by entering the command: `make davinci_xxxx_defconfig`.

Step 2. Choose the driver specific kernel configuration by entering command: `make menuconfig`

Step 3. Select the *Device Drivers* option and then select *Graphics support*. Finally, choose <*> *Support for frame buffer* as a static module and choose <*> *DaVinci Framebuffer support* as a static module.

Step 4. Select the *Device Drivers* option and select *Multimedia devices*. Finally, choose <*> *Video For Linux* as a static module.

Step 5. Select <*> *DaVinci Encoder Manager support* as a static module.

Step 6. Select <*> *DaVinci VPBE support* as a static module for the SD (NTSC/PAL) display via the internal VPBE encoder to COMPOSITE/SVIDEO/COMPONENT outputs.

Step 7. Select <*> *LogicPD Encoder support* as a static module for display via the LogicPD LCD daughtercard to the LogicPD LCD display.

Step 8. Select <*> *THS8200 Encoder support* as a static module for HD (720p/1080i) display via the internal THS8200 daughtercard to COMPONENT1 output.

Step 9. Save your kernel configuration options and build the kernel by entering the following command: `make uImage modules`.

Note: The Linux Open Source community does not recommend building FBDev support as a dynamic module (selecting it as <M>).

3.4 FBDev Driver Configuration

The behavior of the FBDev driver can be configured via a kernel boot argument or using the fbset commands. The display output interface and mode can be selected with either a kernel boot argument or SysFs entries.

Access to the VPBE FBDev Driver is provided through the device entries in the `/dev/fb` directory for OSD and video windows. The following are the device entries for different windows by default:

1. `/dev/fb/0` for the osd0 window.
2. `/dev/fb/1` for the vid0 window.
3. `/dev/fb/2` for the osd1 window.
4. `/dev/fb/3` for the vid1 window.

This mapping of device entries is also listed in the kernel proc file system:

```
$ cat /proc/fb
0 dm_osd0_fb
1 dm_vid0_fb
2 dm_osd1_fb
3 dm_vid1_fb
```

The device number assigned to each device changes with FBDev devices enabled at boot time (see Section 3.4.1.1).

3.4.1 Configure FBDev Display Windows

The VPBE FBDev driver supports the following kernel boot-time command line arguments which you must attach as u-boot bootargs:

video=davincifb

The *video* argument specifies the usage of the FBDev Driver:

- vid0=[off **|** MxNxP, S@X,Y]
- vid1=[off **|** MxNxP, S@X,Y]
- osd0=[MxNxP, S@X,Y]
- osd1=[MxNxP, S@X,Y]

Each argument above (vid0, vid1, osd0, and osd1) defines attributes for the specific display window. They can be concatenated together using a colon (:) sign.

MxN are the horizontal and vertical window size in pixels

P is the color depth in bits per pixel

S is the frame buffer size in bytes with suffix such as K or M for Kilo (2^{10}) or Mega (2^{20})

X, Y are the window coordinates.

The frame buffer size, S, for each video or OSD window defines the maximum frame buffer size reserved by the system at boot time. If any of these attributes are left out, the system sets to its default value. The buffer size can be calculated in the following manner: at 720x480x16 (MxNxP) resolution, a single buffering will be 720x480x2 = 675 Kbytes; 675x2 = 1350 Kbytes, if double buffering is used; 675x3 = 2025 Kbytes, if triple buffering is used. The FBDev driver limits video windows to triple buffering and OSD windows to double buffering. The sum from all frame buffers of these windows is not to exceed 40 Mbytes.

For example, the boot argument for FBDev can be set as:

```
video=davincifb:vid0=720x480x16,2025K@0,0:vid1=720x480x16,2025K@0,0:osd0=720x480x16,1350K@0,0:osd1
=720x480x4,1350K@0,0
```

In the above example, the maximum frame buffer size for vid0 is defined to be in NTSC resolution with 16-bit color depth and is reserved with triple buffer size of 2025K. Its window is positioned at (0,0).

If no FBDev argument is specified, the system sets the display default behavior as:

`dm_osd0_fb:720x480x16@0,0` with buffer size 1350KB

`dm_vid0_fb:0x0x16@0,0` with buffer size 675KB

`dm_osd1_fb:720x480x4@0,0` with buffer size 1350KB

`dm_vid1_fb:0x0x16@0,0` with buffer size 675KB

3.4.1.1 Disable FBDev Video Windows at Boot Time

A specific FBDev video window can be disabled using the boot argument option:

```
video=davincifb:vid0=off:vid1=off
```

or

```
video=davincifb:vid0=0,0:vid1=0,0
```

In this example, both the vid0 and vid1 are disabled at boot time. This prevents the FBDev driver from creating devices for vid0 and vid1 and the device mapping is rearranged to have only two entries.

```
$ cat /proc/fb
0 dm_osd0_fb
1 dm_osd1_fb
```

Note that `/dev/fb/2` and `/dev/fb/3` no longer exist in the system.

If any of the windows are disabled, any FBDev driver application is not allowed to perform any I/O control operation with that window. In the above case, however, this allows V4L2 applications to access the video devices. Note that OSD windows cannot be turned off by boot arguments since FBDev is the only video driver in LSP that can access OSD windows. Therefore, even if setting it up as *off* value in the boot arguments, it is ignored by the FBDev driver and set up with default values.

3.4.1.2 Enable FBDev Video Windows at Boot Time Without FBDev Driver Claiming the Windows

Alternatively, boot arguments can be used to prevent the FBDev driver from claiming video windows while still reserving the frame buffer space and creating FBDev devices. In other words, this allows V4L2 applications to access vid0 and vid1 windows, yet FBDev devices `/dev/fb/1` and `/dev/fb/3` are still created.

```
video=davincifb:vid0=0,2025K:vid1=0,1350K
```

After booting up, all FBDev applications are created as normal, and V4L2 applications are able to claim video windows (through `/dev/video/2` or `/dev/video3`) to use. When an FBDev application needs to use the device, you need to use the `fbset` command which allows the FBDev driver to re-claim the video windows (to desired resolution):

```
$ fbset -fb /dev/fb/1 -xres 720 -yres 480 -vxres 720 -vyres 1440 -depth 16
$ fbset -fb /dev/fb/3 -xres 720 -yres 480 -vxres 720 -vyres 1440 -depth 16
```

3.4.1.3 Release of FBDev Display Windows after Boot Time

Instead of disabling windows using boot arguments, you can use fbset to release the windows from the FBDev driver for others to use even if the FBDev devices are enabled. The following example shows the commands to *turn off* osd0 and vid0 windows, respectively:

```
$ fbset -fb /dev/fb/0 -xres 0
$ fbset -fb /dev/fb/1 -xres 0
```

When these display windows need to be used by an FBDev application, use the `fbset` command, similar to those in Section 3.4.1.2, again to restore the frame buffer device.

3.4.2 Configure Display Output and Mode

In LSP1.20, output interface parameters you set are passed to the Encoder Manager for processing. They can be set through the Encoder Manager boot argument at boot time as follows:

```
“davinci_enc_mngr.ch0_output=COMPOSITE davinci_enc_mngr.ch0_mode=NTSC”
```

Or, after the kernel boots up, they can be set by writing the output string and mode string into two DaVinci SysFs attributes:

```
/sys/class/davinci_dsplay/ch0/output
/sys/class/davinci_display/ch0/mode
```

For details of set up and supported output and mode strings, see the *LSP 1.20 DaVinci Video Sysfs User's Guide* (SPRUEL6).

4 User Application Interfaces

4.1 API Classification

This section introduces the application-programming interface for the FBDev Driver by grouping them into logical units.

4.1.1 Configuration

This section contains FBDev Driver APIs that give you the ability to specify the desired configuration parameters. IOCTLs like FBIO_SET_VID_CONFIG_PARAMS help to customize the device driver parameters. Section 4.3.2 elaborates on each such mechanism in greater detail.

4.1.2 Initialization

This section contains the FBDev Driver APIs that are intended for use in component initialization.

The API open is used for initializing of the VPBE driver.

4.1.3 Memory Mapping Considerations and Buffer Programming

Applications have to perform a memory map (mmap) of the video buffer before using it to either display a static image or perform buffer-pointer flipping operations using multiple buffers. The buffer organization of the video buffer is as shown in Figure 6. The extra padding in the x-direction is needed due to some hardware restrictions on the offset value which has to be a multiple of 32 bytes. The following is using DM644x system as an example:

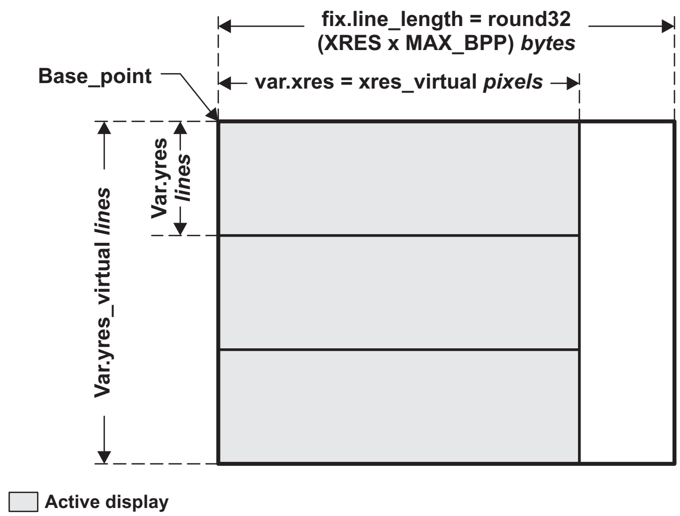
The buffer organization takes into account the highest resolution in x-direction that is supported by the driver, which is 1920, and the maximum bpp that is supported by the respective windows, in this case, OSDs = 16(RGB565) and VID = 24(RGB888).

In effect, the line_length for the respective windows becomes:

OSD0 = OSD1 = round32 (1920 * 2) = 3840

VID0 = VID1 = round32 (720 * 3) = 5760

Figure 6. Buffer Organization



If the applications want to map the entire buffer, the amount to be mapped is:

size = fix.linelength * var.yres_virtual.

If the applications want to map only one buffer of the display memory, the amount to be mapped is:

size = fix.linelength * var.yres

When writing to subsequent buffers, the following equation can be applied. Writing to the nth buffer (n ranging from 0..2 or 3):

address=base_pointer + fix.linelength * var.yres * (n % 3)

This address is passed as part of FBIOPAN_DISPLAY ioctl for driver to program the VPBE register.

4.1.4 Control

This section contains the FBDev Driver APIs that are intended for use in controlling the functioning of the driver during run-time. For example, you can use the IOCTL FBIOBLANK to enable/disable the window.

4.1.5 Data Acquisition/Processing

This section contains the list of the driver APIs that help you to output parameters out of the driver.

IOCTLs like FBIO_GET_VIDEO_CONFIG_PARAMS are used to get video parameters the hardware.

The IOCTL FBIO_GET_BITMAP_CONFIG_PARAMS is used to get bitmap configuration.

4.1.6 Termination

This section contains the driver APIs that help in gracefully terminating the deployed driver run-time entities.

The API close is used to free all the resources that are being acquired at time of initialization and creation.

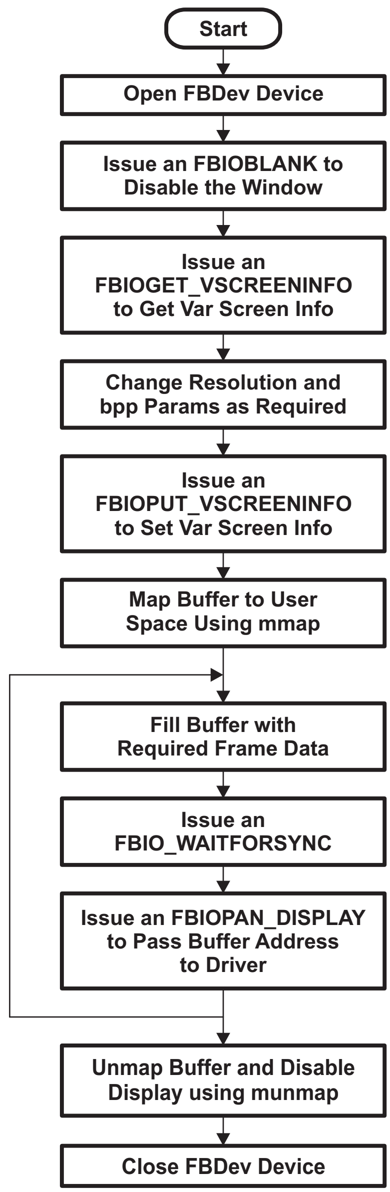
4.2 API Usage Scenarios Example

Figure 7 shows the simple usage scenarios of the DaVinci FBDev Driver API for an application that streams multiple frames onto a video window. Note the loop in this flowchart:

- An FBDev application usually employs more than one display buffer (usually 3). One is for the driver to use and the other two are for frame buffering. The application always fills out the next frame(s) while the driver is displaying the current frame.
- After filling the frame is done, the application uses the FBIO_WAITFORVSYNC ioctl to wait for the next end-of-frame interrupt.
- Upon receiving the interrupt, the application returns from the FBIO_WAITFORVSYNC ioctl call and immediately calls the FBIOPAN_DISPLAY ioctl to have the driver pass on the address of the frame base address to the VPBE register for the hardware to display.

For further details of using these function APIs and ioctls, see Section 4.3 and the loopback example code as part of the release. Most API usage can also be found in other example codes.

Figure 7. VPBE Functional Flow Diagram



4.3 API Specification

4.3.1 Naming Conventions

The naming conventions are followed as per the Linux Standard.

4.3.2 DaVinci VPBE Device Driver Functions

The detailed descriptions of APIs discussed above are described below, in alphabetical order.

API close

Prototype `int close(int fd)`

Description Closes the logic channel associated with fd.

Arguments

Arg1	int fd
Arg2	NA
Arg3	NA

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `close(fd);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_ENABLE_DISABLE_WIN (Deprecated)

Prototype `int ioctl(int fd, int request, unsigned char enable_flag)`

Description The driver enables the specified window or disables it. No memory allocation or freeing is done here.

This ioctl is deprecated. Use FBIOBLANK, instead, or use FBIOPUT_VSCREENINFO with any xres, yres, xres_virtual, yres_virtual set to zero, then the window is disabled. It can be reenabled by using FBIOPUT_VSCREENINFO again and setting a valid video mode.

Arguments

Arg1	int fd
Arg2	int request
Arg3	unsigned char enable_flag

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example

```
ioctl(fd, FBIO_ENABLE_DISABLE_WIN, enable_flag);
```

Side Effects None

See Also None

Errors None

IOCTL FBIO_SET_BITMAP_BLEND_FACTOR

Prototype

```
int ioctl(int fd, int request, struct vpbe_bitmap_blend_params* argp)
```

Description

Used for setting the blend factor for the bitmap window. The amount of blending (i.e., relative amount of video data vs. bitmap data) at each pixel is determined by the blending factor.

The OSD also supports transparency blending mode. If transparency is enabled, any pixel on the bitmap display that has a value of 0 will be transparent (or partially transparent) and allow the underlying video pixel to be displayed based on the blending factor.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_bitmap_blend_params* argp

Return Value

Zero, on success, or -1, if an error occurred.

Calling Constraints

Blending is supported only for bitmap windows.

When color keying is disabled, the entire bitmap window is blended with the video windows according to blending factor.

When color keying is enabled in bitmap mode, blending is only performed for pixels whose bitmap value is 0, according to blending factor.

When color keying is enabled in RGB mode and the pixel value is the same as the color key, the YCbCr data converted from the RGB value and video windows are blended according to the blending ratio specified by blending factor.

Valid values for the blend factor are 0 to 7. Any other value causes the invalid parameter value error.

Example

```
ioctl(fd, FBIO_SET_BITMAP_BLEND_FACTOR, & bitmap_blend_params);
```

Side Effects

None

See Also

None

Errors

None

IOCTL FBIO_SET_BITMAP_WIN_RAM_CLUT (Deprecated)

Prototype

```
int ioctl(int fd, int request, unsigned char* argp)
```

Description

Two bitmap windows are supported on the VPBE that allow the display of graphics and icons. The bitmap window uses a color look-up table (CLUT), either in ROM or RAM, to determine the actual display color for a given bitmap pixel value. A total of 256 CLUT entries, in YUV422 color space are used. This ioctl is used to set up the color look-up table in RAM as per user specifications. The color value of each of the 256 pixels is specified using the following array:

```
unsigned char  ram_clut[256][3];
```

You must initialize the two-dimensional array of 256 color values with [0] as luma, [1] ChromaCb, and [2] ChromaCr and pass the pointer to this array as the parameter.

This ioctl is deprecated. Use FBIOPUTCMAP with RGB format, the FBDev driver converts it to YUV format internally.

Arguments

Arg1	int fd
Arg2	int request
Arg3	unsigned char* argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example

```
ioctl(fd, FBIO_SET_BITMAP_WIN_RAM_CLUT, ram_clut);
```

Side Effects None

See Also None

Errors None

IOCTL FBIO_ENABLE_DISABLE_ATTRIBUTE_WIN (Deprecated)

Prototype `int ioctl(int fd, int request, unsigned char enable_flag)`

Description

Only OSD1 can be configured as an attribute window instead of a bitmap window. In this mode, the attribute window allows blending and blinking on a pixel-by-pixel basis in bitmap window 0. The value of enable_flag turns the function on or off (0 = off, 1 = on).

The enabled/disabled status of OSD1 is unchanged by this ioctl. To avoid display glitches, you should disable OSD1 prior to calling this ioctl.

When enabling attribute mode, var→bits_per_pixel is set to 4; var→xres, var→yres, var→xres_virtual, var→yres_virtual, win→xpos, and win→ypos are all copied from OSD0. var→xoffset and var→yoffset are set to 0. No changes are made to the OSD1 configuration if OSD1 is already in attribute mode.

When disabling attribute mode, the window geometry is unchanged; var→bits_per_pixel remains set to 4. No changes are made to the OSD1 configuration if OSD1 is not in attribute mode.

Arguments

Arg1	int fd
Arg2	int request
Arg3	unsigned char enable_flag

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints Setting the attribute window or the bitmap window can be done only when the window is disabled.

Example

```
ioctl(fd, FBIO_ENABLE_DISABLE_ATTRIBUTE_WIN, enable_flag);
```

Side Effects None

See Also None

Errors None

IOCTL FBIO_GET_BLINK_INTERVAL

Prototype `int ioctl(int fd, int request, struct vpbe_blink_option* argp)`

Description Used to get the existing blinking interval of the attribute window and value of the vpbe_blink_enable flag.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_blink_option* argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_GET_BLINK_INTERVAL, blink_enable_flag);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SET_BLINK_INTERVAL

Prototype `int ioctl(int fd, int request, struct vpbe_blink_option* argp)`

Description Used to set the blinking of the attribute window. The blinking interval can be set in the structure.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_blink_option* argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SET_BLINK_INTERVAL, &blink_option);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_GET_VIDEO_CONFIG_PARAMS

Prototype `int ioctl(int fd, int request, struct vpbe_video_config_params * argp)`

Description Used to get the existing configurations of the video window. Configuration parameters are listed in the structure below.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_video_config_params * argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_GET_VIDEO_CONFIG_PARAMS, &video_config_params);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SET_VIDEO_CONFIG_PARAMS

Prototype `int ioctl(int fd, int request, struct vpbe_video_config_params * argp)`

Description Used to set the configurations of the video window. Configuration parameters are listed in the structure.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_video_config_params * argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SET_VIDEO_CONFIG_PARAMS, &video_config_params);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_GET_BITMAP_CONFIG_PARAMS

Prototype `int ioctl(int fd, int request, struct vpbe_bitmap_config_params* argp)`

Description Used to get the existing configurations of the bitmap (OSD0/OSD1) window. Configuration parameters are listed in the structure below.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_bitmap_config_params* argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_GET_BITMAP_CONFIG_PARAMS, & bitmap_config_params);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SET_BITMAP_CONFIG_PARAMS

Prototype `int ioctl(int fd, int request, struct vpbe_bitmap_config_params* argp)`

Description Used to set the configurations of the BITMAP (OSD0/OSD1) window. Configuration parameters are listed in the structure below.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_bitmap_config_params* argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SET_BITMAP_CONFIG_PARAMS, & bitmap_config_params);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SET_BACKG_COLOR

Prototype `int ioctl(int fd, int request, struct vpbe_backg_color *argp)`

Description Used to set the background color. The window should be disabled before using this IOCTL.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_backg_color *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SET_BACKG_COLOR, &backg_color);`

Side Effects None

See Also None

Errors None

IOCTL FBIOGET_VSCREENINFO

Prototype `int ioctl(int fd, int request, struct fb_var_screeninfo *argp)`

Description Used to get the variable screen information of the frame buffer. For each frame buffer window, this ioctl is used to get the var info.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_var_screeninfo *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `Ioctl(fd, FBIOGET_VSCREENINFO, &var_info);`

Side Effects None

See Also None

Errors None

IOCTL FBIOPUT_VSCREENINFO

Prototype `int ioctl(int fd, int request, struct fb_var_screeninfo *argp)`

Description Used to set variable screen parameters for the frame buffer which include:

- Window resolution.
- Bits per pixel for the window. This sets the input format of the window.
- Validation for the all VPBE rules is done; if these rules are violated, then an error is returned.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_var_screeninfo *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints This ioctl can be called when the window is disabled or enabled.

Values of xres, yres, and bpp can be changed. It is assumed that for a bpp of 24, RGB888 is set and for a bpp of 16, RGB565 is set.

For VID windows numbufs is 3; whereas, for bitmap windows numbufs is 2.

Example `ioctl(fd, FBIOPUT_VSCREENINFO, &var_info);`

Side Effects None

See Also None

Errors None

IOCTL FBIOPUTCMAP

Prototype `int ioctl(int fd, int request, struct fb_cmap_user *argp)`

Description Used to set up a pseudo palette.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_cmap_user *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIOPUTCMAP, &cmap_user);`

Side Effects None

See Also None

Errors None

IOCTL FBIOGETCMAP

Prototype `int ioctl(int fd, int request, struct fb_cmap_user *argp)`

Description Returns the cmap to the application.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_cmap_user *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIOGETCMAP, &cmap_user);`

Side Effects None

See Also None

Errors None

IOCTL FBIOPAN_DISPLAY

Prototype `int ioctl(int fd, int request, struct fb_var_screeninfo *argp)`

Description Used to set the window display buffer. Using the var_screeninfo, the offset of the buffer (out of the number of buffers for the window, 3 for video and 2 for osd) is passed to the ioctl. The actual buffer location is calculated and set in the window register.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_var_screeninfo *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_PANDISPLAY, &var_info);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SET_CURSOR

Prototype `int ioctl(int fd, int request, struct fb_cursor * argp)`

Description Used to configure cursor parameters.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_cursor * argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints fg_color always points to the ROM CLUT. Cursor horizontal and vertical line width are equal and set to depth.

This IOCTL can be called by the vid0 file descriptor only.

Example `ioctl(fd, FBIO_SET_CURSOR, &cursor);`

Side Effects None

See Also None

Errors None

IOCTL FBIOGET_CON2FBMAP

Prototype `int ioctl(int fd, int request, struct fb_con2fbmap *argp)`

Description Gets the con2fbmap structure.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_con2fbmap *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIOGET_CON2FBMAP, &con2fbmap);`

Side Effects None

See Also None

Errors None

IOCTL FBIOSET_CON2FBMAP

Prototype `int ioctl(int fd, int request, struct fb_con2fbmap *argp)`

Description Sets the con2fbmap structure.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_con2fbmap *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIOSET_CON2FBMAP, &con2fbmap);`

Side Effects None

See Also None

Errors None

IOCTL FBIOBLANK

Prototype `int ioctl(int fd, int request, int enable)`

Description Used to disable the entire display window.

Arguments

Arg1	int fd (the window to be enabled/disabled)
Arg2	int request
Arg3	int enable flag

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `Ioctl(fd, FBIOBLANK, 1)`

Side Effects None

See Also None

Errors None

IOCTL FBIO_WAITFORSYNC

Prototype `int ioctl(int fd, int request)`

Description Used to wait until the frame is displayed; it returns when the vsync event is received.

Arguments

Arg1	int fd
Arg2	int request
Arg3	None

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_VAITFORSYNC);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SETATTRIBUTE (Deprecated)

Prototype `int ioctl(int fd, int request, struct fb_fillrect *argp)`

Description Used to fill up attribute values 0-7 using the fb_fillrect structure.

This ioctl is deprecated. The application writes attribute pixel values directly to the rectangular area in the OSD1 frame buffer.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct fb_fillrect *argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SETATTRIBUTE, &fillrect);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SETPOS

Prototype `int ioctl(int fd, int request, struct vpbe_window_position * argp)`

Description Used to set up window position.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct vpbe_window_position * argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SETPOS, &window_position);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SETZOOM

Prototype `int ioctl(int fd, int request, struct zoom_params* argp)`

Description Used to set the display window to zoom by 2x or 4x.

Arguments

Arg1	int fd
Arg2	int request
Arg3	struct zoom_params* argp (window id is ignored)

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SETZOOM, & zoom_params);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SETPOSX

Prototype `int ioctl(int fd, int request, int* argp)`

Description Used to program the start position of the planes in X direction,

Arguments

Arg1	int fd
Arg2	int request
Arg3	int * argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SETPOSX, 32);`

Side Effects None

See Also None

Errors None

IOCTL FBIO_SETPOSY

Prototype `int ioctl(int fd, int request, int  * argp)`

Description Used to program the start position of the planes in Y direction.

Arguments

Arg1	int fd
Arg2	int request
Arg3	int * argp

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `ioctl(fd, FBIO_SETPOSY, 40);`

Side Effects None

See Also None

Errors None

API mmap

Prototype `void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)`

Description Maps the frame buffers allocated by the davincifb module in kernel space to user space. Window size and input format IOCTLs define the size of the memory required. Whereas the enable window ioctl actually allocates the memory, this function maps the allocated memory buffers to the user space.

Arguments

Arg1	void *start
Arg2	size_t length
Arg3	int prot
Arg4	int flags; Only MAP_SHARED is supported
Arg5	int fd
Arg6	off_t offset

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `mmap(0,image_size, MAP_SHARED, fd, offset);`

Side Effects None

See Also None

Errors None

API munmap

Prototype `int munmap(void *start, int length)`

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

Arguments

Arg1	void *start
Arg2	size_t length
Arg3	NA

Return Value Zero, on success, or -1, if an error occurred.

Calling Constraints None

Example `munmap(offset, image_size)`

Side Effects None

See Also None

Errors None

API open

Prototype `int open(char *name, int mode)`

Description Opens an instance of the frame buffer device. For every window supported by the VPBE subsystem, a separate frame buffer device is opened.

Arguments

Arg1	char *name The values are: • `/dev/fb/0` for OSD0 window • `/dev/fb/1` for VID0 window • `/dev/fb/2` for OSD1 window • `/dev/fb/3` for VID1 window
Arg2	int mode (Only O_RDWR is supported)
Arg3	NA

Return Value File descriptor, on success, or -1, if an error occurred.

Calling Constraints None

Example `open("/dev/fb/0", O_RDWR)`

Side Effects None

See Also None

Errors None

4.4 Supported Display Formats

Table 2 shows the supported formats for FBDev display windows.

Table 2. Supported Formats for FBDev Display Windows

Device	VID0	VID1	OSD0 [(1)]	OSD1 [(1)]
DM355	YUV422	YUV422	RGB565 (16-bit), RGB888 (32-bit), YUV422, Bitmap (1/2/4/8)	RGB565 (16-bit), RGB888 (32-bit), YUV422, Bitmap (1/2/4/8) or Attribute (Blend + Blink)
DM6446 [(2)]	YUV422/RGB888 (24-bit)	YUV422, RGB888 (24-bit)	RGB565 (16-bit), Bitmap (1/2/4/8)	RGB565 (16-bit), Bitmap (1/2/4/8) or Attribute (Blend + Blink)

(1) OSD0 and OSD1 cannot both be in RGB565/RGB888 or YUV mode at the same time, the other window must be in bitmap mode or attribute mode.
(2) For DM6446, VID0 and VID1 cannot both be in RGB888 mode at the same time.

4.5 DM355 OSD Constraints with HD Mode

When supporting HD modes (720p/1080i), the DM355 OSD window has the following constraint: When OSD1 is configured to be an attribute window and OSD0 is configured to accept RGB565 data, random noise is seen on the OSD0 window. This noise happens irrespective of the OSD0 window size.

This issue is believed to be caused by a hardware bandwidth limitation and is currently under investigation.

To display HD, the following configuration for display windows has been tested with 1080i mode without seeing any noise on OSD windows:

- OSD0 - 720x480 (bitmap - 8)
- OSD1- 720X480 (bitmap - 8)
- VID0 - 1920x1080 (UYVY)

4.6 DM6446 HD Constraints

For a list of HD constraints, see the *Video Window Constraints* section of the *TMS320DM644x DMSoC Video Processing Back End (VPBE) User's Guide* (SPRUE37).

4.7 Use fbset Command to Configure Display Windows

In addition to the usage introduced in Section 3.4, the `fbset` command can be used by an application (or at console) to configure the frame buffer dimensions. The following are some examples:

- Make OSD0 720x480x16 (RGB565) with double buffering:

```
$ fbset -fb /dev/fb/0 -xres 720 -yres 480 -vxres 720 -vyres 960 -depth 16 -nonstd 0
```

- Make OSD0 720x480x32 (RGB888) with double buffering on DM355:

```
$ fbset -fb /dev/fb/0 -xres 720 -yres 480 -vxres 720 -vyres 960 -depth 32 -nonstd 0
```

- Make OSD0 640x480x32 (RGB888) with double buffering on DM355 using a VGA LCD display (progressive):

```
$ fbset -fb /dev/fb/0 -xres 640 -yres 480 -vxres 640 -vyres 960 -depth 32 -nonstd 0 -laced 0
```

- Make OSD0 720x480x8 (8-bpp Bitmap) with double buffering:

```
$ fbset -fb /dev/fb/0 -xres 720 -yres 480 -vxres 720 -vyres 960 -depth 8 -nonstd 0
```

- Make VID0 720x480x16 (YCbCr) with triple buffering:

```
$ fbset -fb /dev/fb/1 -xres 720 -yres 480 -vxres 720 -vyres 1440 -depth 16 -nonstd 1
```

- Make OSD1 720x480x4 (attribute mode) with double buffering:

```
$ fbset -fb /dev/fb/2 -xres 720 -yres 480 -vxres 720 -vyres 960 -depth 4 -nonstd 1
```

- Make VID1 720x480x16 (YCbCr) with triple buffering:

```
$ fbset -fb /dev/fb/3 -xres 720 -yres 480 -vxres 720 -vyres 1440 -depth 16 -nonstd 1
```

The *-nonstd* (non-standard) option sets the non-standard mode window. If the *-nonstd* option is set to non-zero, it is to indicate the use of pixel format other than RGB/bitmap mode. To set a window in YUV mode, *-nonstd* needs to be non-zero; to set an OSD1 window to be an attribute window, this option also needs to be non-zero.

The *-laced* (interlaced) option specifies whether the display is interlaced or not. Non-zero means interlaced (the default value if *-laced* is not specified).

Note that, by default, the FBDev driver always makes the initial window resolution (xres, yres) and frame buffer resolution (vxres, vyres) match. Therefore, in order to utilize the extra frame buffer allocated at boot time (Section 3.4.1), use the `fbset` command to explicitly set virtual vertical resolution to the desired buffering size to support frame flipping, as shown in the above examples.

To check the current configuration of a specific FBDev display window, use
`$ fbset -I -fb /dev/fb/[0,1,2,3]`.

4.8 FBDev Driver Function Hooks

Table 3 lists the DaVinci routines (from `davincifb.c`) that are implemented for FBDev function hooks (from `fb.h`) in this release and their usages. For the hooks that are not listed here, they are either generic (from `fbmem.c`) or not used (not defined).

Table 3. FBDev Driver Functions

FBDev Function Hooks in fb.h	DaVinci Functions in davincifb.c	Usage	Remarks
fb_check_var	davincifb_check_var	Validate fb_var_screeninfo values	
fb_set_par	davinci_fb_set_par	Set/change fb_var_screeninfo values	
fb_setcolreg	davincifb_setcolreg	Set color register	
fb_blank	dacincifb_blank	FBIOBLANK	
fb_pan_display	davincifb_pan_display	FBIOPAN_DISPLAY	
fb_ioctl	davincifb_ioctl	fb_ioctl (standard ioctl-processing routine)	To execute DaVinci-specific ioctl functions.

4.9 Example Applications

The following sample applications are provided to showcase display functionality of the FBDev Display as part of the release. They do loopback of captured video to the display for the modes listed below:

- NTSC
- PAL
- 480P-60
- 576P-50
- 640x480

For usage details on the FBDev loopback application, refer to
`PSP_##_##_##_###/examples/fbdev/readme.txt`.

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265