![Texas Instruments logo] **TEXAS INSTRUMENTS**

*Application Report*
*SPRAAN0−June 2007*

# Digital Video Using DaVinci SoC

*Juan Gonzales, Neal Frager, Ryan Link*

**ABSTRACT**

This application report provides a brief video basics overview. The Texas Instruments (TI) TMS320DM6446/DM6443 DaVinci™ system-on-chip (SoC) devices are capable of YCbCr or red, green, blue (RGB) digital video output; YCbCr outputs can interface with a video encoder or RGB outputs can interface with LCD parts. This document details these two solutions, and provides software drivers and an example application to demonstrate these solutions.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www-s.ti.com/sc/techlit/spraan0.zip. See the Readme_first.txt file included with the download for information about software compatibility.

**Contents**

**List of Figures**

**List of Tables**

# 1    Video Standards

The digital LCD controller can be programmed to conform to high-definition television (HDTV) standards, such as SMPTE274M (1080i) and SMPTE296M (720p). These HDTV standards follow 4:2:2 conventions where luminance and chrominance signals are separated on to two different 8- or 10-bit data paths. The THS8200 receives this YCbCr 4:2:2 data and produces the corresponding YPbPr analog representation that is widely supported by HD displays.

The digital LCD controller can also be programmed to output RGB video, which is commonly accepted by computer monitors and LCD displays. For digital LCDs, the term video graphics array (VGA) standard describes a display with a 640×480 pixel resolution with a frame rate of 60 frames per second (fps) and 16 bits per pixel. Logic Product Development's RGB666 LCD Kit can connect TMS320DM644x devices to digital LCD displays.

## 1.1    RGB Pixel Format

Before presenting the Y/C 4:2:2 format, first consider the RGB pixel format as a camera sensor and how the displays normally work in RGB color space. Figure 1 is a typical representation of an RGB pixel in a hardware data bus.
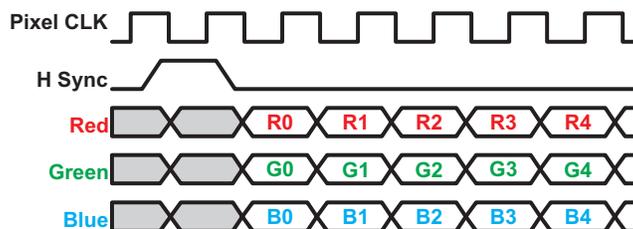


**Figure 1. RGB Pixel Representation**

---

DaVinci is a trademark of Texas Instruments.
Linux is a registered trademark of Linux Torvalds in the U.S. and other countries.

---

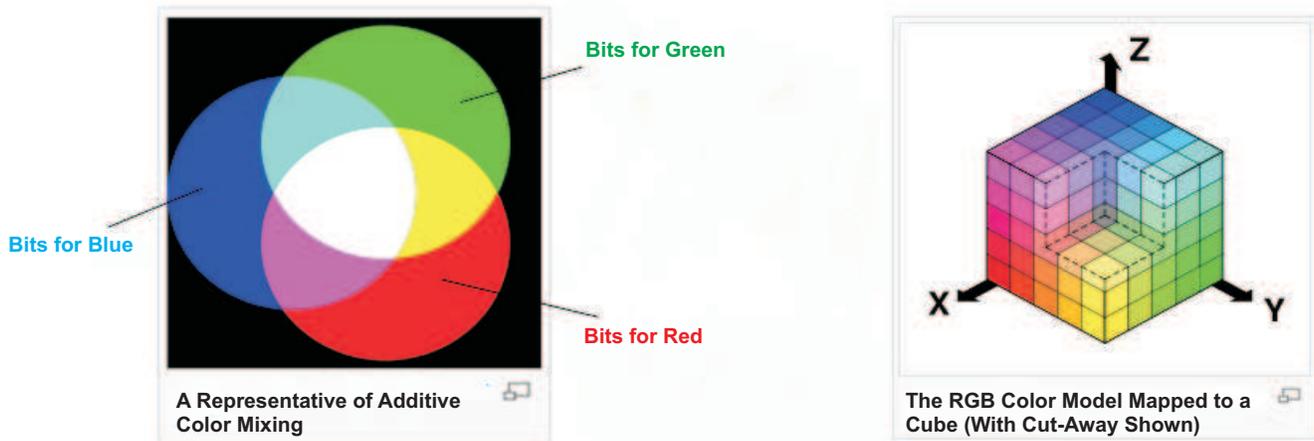Essentially, each video pixel contains a component of red, green, and blue video data. True color can be obtained through the blending of these primary colors as shown in Figure 2.

**Pixels Split Into Color Components**



A Representative of Additive Color Mixing

The RGB Color Model Mapped to a Cube (With Cut-Away Shown)

**Figure 2. RGB Color Components**

There are many common standards for RGB pixel data, varying from RGB332 up to RGB888. The numbers for each standard correspond to the number of bits allocated for each color component of video data. For example, RGB332 is a common 8-bit pixel standard that has 3 bits of red, 3 bits of green, and 2 bits of blue data for each pixel. RGB565 is a common 16-bit standard with 5 bits of red, 6 bits of green, and 5 bits of blue. Finally, RGB888 is a 24-bit standard with 8 bits of each color component.

As the number of bits per pixel increases, the levels of blending and color depth increase exponentially. The greater the bits per pixel, the greater the color depth and quality, but at the same time, the greater the amount of space required to store each image of the video stream.

## 1.2  Y/C Color Mode and 4:2:2 Pixel Format

One of the advantages of YCbCr color space over RGB is Y, or the luma component, that represents the brightness that transmits signals to black and white displays, whereas the U and V, or chroma components, represent the color of the picture. Figure 3 shows an example of YCbCR color space.



Luma (Y)

Chroma Blue (Cb)

Chroma Red (Cr)
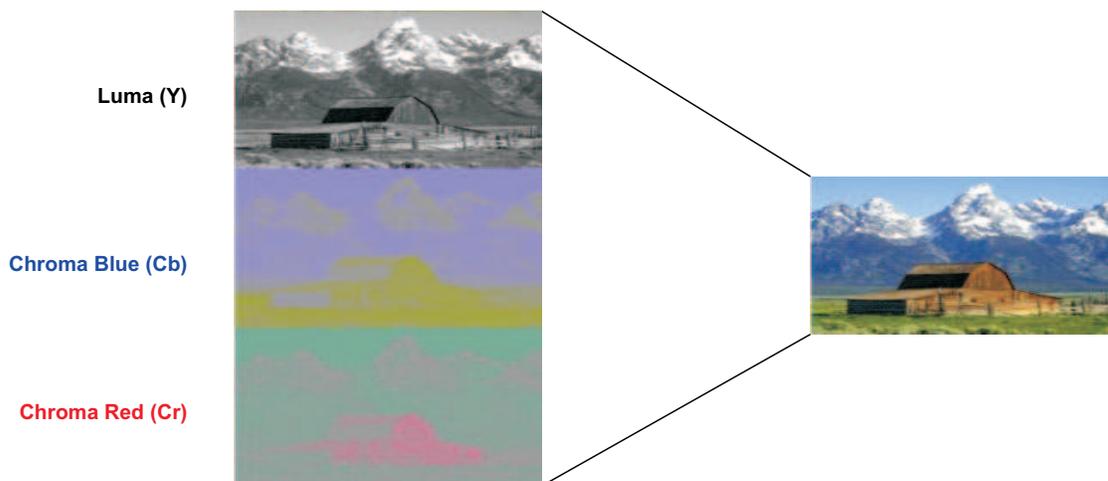
**Figure 3. YCbCr Color Space**

In addition, the human eye is more sensitive to the Y component than to the Cb and Cr components. As seen from the image in Figure 3, you can easily distinguish that the image is a barn with mountains in the background by looking at the Y, or luma component, alone. However, with the chroma blue and chroma red components alone, it is difficult to recognize the image. Taking this feature into account, you can safely remove some of the chroma data and save on both bandwidth and disk space. Variations of the YCbCr pixel representations exploit this feature (for instance, YCbCr 4:4:4, 4:2:2, and 4:2:0).
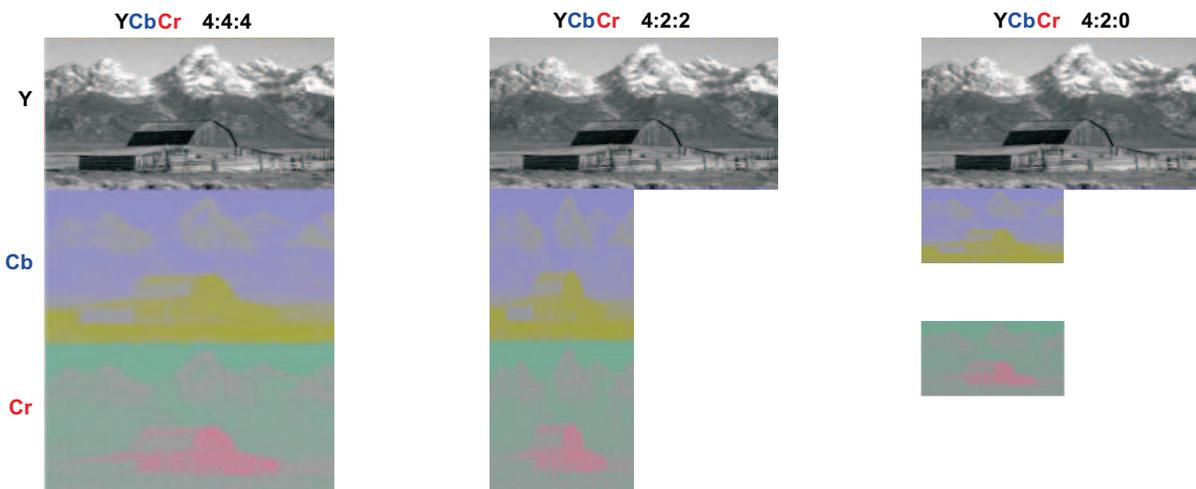


**Figure 4. Various YCbCr Pixel Representations**

Figure 4 demonstrates the different ratios of luma data to chroma data that exist in some of the most common Y/C standards. In the YCbCr 4:4:4 case, the ratio of Y luma data to each of the chroma components is 4:4 or 1:1. In the case of YCbCr 4:2:2, the amount of chroma data per line is reduced by half. YCbCr 4:2:0 takes this one step further by additionally reducing the chroma data by half in the vertical space as well. Since it is difficult to distinguish between an image made with each of these three standards, it would make sense for developers to always use YCbCr 4:2:0 whenever they can. However, other formats such as the DaVinci video drivers that go with YCbCr 4:2:2 are also useful.

Observe how the different YCbCr pixel formats typically appear in a hardware bus, which explains why the Texas Instruments DaVinci products use YCbCr 4:2:2 as the standard of choice for digital video ports.

Figure 5 depicts an YCbCr 4:4:4 data bus. Note that this bus appears very similar to the RGB data bus. As with RGB, YCbCr 4:4:4 requires a bus for each of the three components.
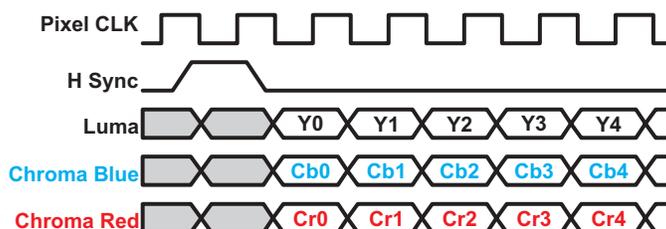


**Figure 5. YCbCr 4:4:4 Pixel Representation**

If some of the chroma data is removed, the benefits become clear (Figure 6). Notice that 4:2:2 pixels only require two data buses instead of three, as the Cb and Cr busses are multiplexed and share a common bus (very typical in hardware).



**Figure 6. YCbCr 4:2:2 Pixel Representation**

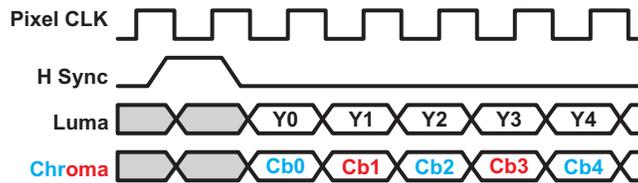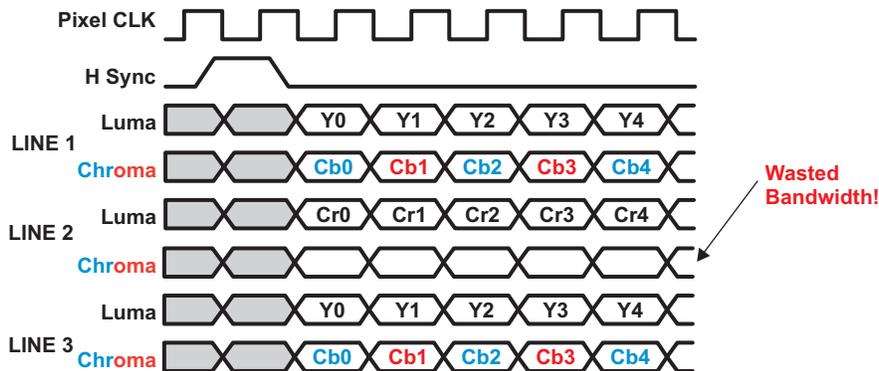However, if more chroma data is removed, as seen in Figure 7, you can see that there are no additional benefits from a hardware data bus standpoint.



**Figure 7. YCbCr 4:2:0 Pixel Representation**

Thus, YCbCr 4:2:2 is a very popular pixel format used in the digital domain, and is supported by the video ports on all Texas Instruments DaVinci processors.

## 1.3 SMPTE296M

SMPTE296M describes the scanning and timing information required for a 1280×720 resolution with progressive scanning. This standard is commonly known as 720p. Frame rates within this standard can vary from 23.976 f/s to 60 f/s. The sampling frequency, or pixel clock, stays fixed at ~74.25 MHz. Due to color and sound sub-carrier frequency interactions, pixel clock rates may need to be modified by a 1.001 ratio, which decreases the pixel clock rate from 74.25 MHz to 74.176 MHz. To achieve various frame rates, the number of pixels per line varies accordingly; however, the number of actual lines remains constant at 750 lines per frame. Table 1 provides a listing for common 720p resolutions. The more common mode for this standard of broadcast television used in the USA is 1280×720×30 fps; this is used in the project and discussed later in the document. Both luminance and chrominance data are transmitted using Y/C mode and 4:2:2 pixel format. Figure 8 illustrates the vertical and horizontal components of SMPTE296M.
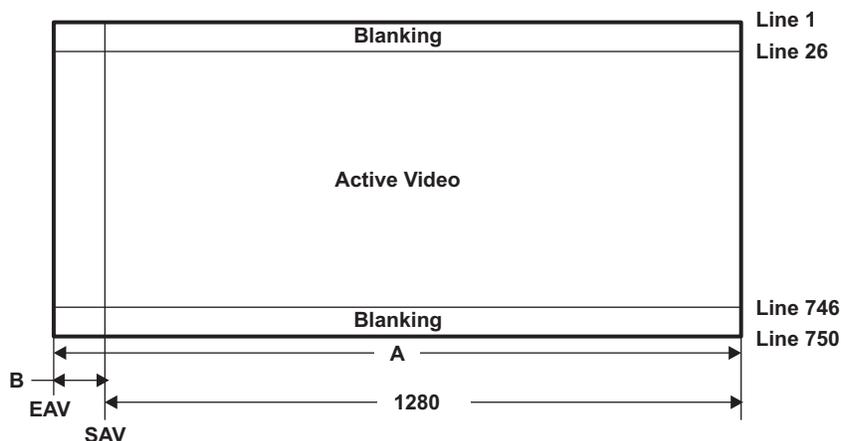
**Figure 8. SMPTE296M Vertical and Horizontal Components**

See Table 1 for A and B pixel counts. Lines 26 – 746 make up the 720 horizontal lines of active video. The 1280 vertical lines of active video are defined by the total horizontal resolution (A) minus the horizontal blanking count (B).

**Table 1. Various SMPTE296M Frame Rates**

| Active Horizontal Resolution | Frame Rate (Hz) | Sample Rate/ Pixel Clock (MHz) | Total Horizontal Resolution (A) | Horizontal Blanking (B) |
|---|---|---|---|---|
| 1280 | 23.976 | 74.176 | 4125 | 2845 |
| 1280 | 24 | 74.25 | 4125 | 2845 |
| 1280 | 25 | 74.25 | 3960 | 2680 |
| 1280 | 29.97 | 74.176 | 3300 | 2020 |
| 1280 | 30 | 74.25 | 3300 | 2020 |
| 1280 | 50 | 74.25 | 1980 | 700 |
| 1280 | 56.94 | 74.176 | 1650 | 370 |
| 1280 | 60 | 74.25 | 1650 | 370 |

## 1.4 SMPTE274M

SMPTE274M varies slightly from SMPTE296M. The main difference between the two standards is the resolution. SMPTE274M describes the resolution and timing information required for 1920×1080 resolutions (among others). Table 2 and Table 3 provide listings of the various frame rates, pixel clocks, and resolutions for SMPTE274M, interlaced and progressive formats, respectively. The more common mode for this standard of broadcast television used in the USA is 1920×1080×30 fps interlaced, commonly known as 1080i; this is what the project uses and is discussed later in the document. Similar to SMPTE296M, frame rates vary from 23.976 f/s to 60 f/s. The total number of lines remains constant at 1125. The number of samples per line varies to achieve desired frame rates. Pixel clock rates double when high frame rates via progressive mode are desired. Y/C mode and 4:2:2 pixel format are also used with the SMPTE274M standard. Figure 9 illustrates the vertical and horizontal components of SMPTE274M.

**Figure 9. SMPTE274M Interlaced Vertical and Horizontal Components**

See Table 2 for A and B pixel counts. Lines 21 – 561 make up odd numbered horizontal lines of active video. Lines 584 – 1124 make up the even horizontal lines of active video. The vertical lines of active video are defined by the total horizontal resolution (A) minus the horizontal blanking count (B).

**Table 2. Various SMPTE274M Interlaced Frame Rates**

| Active Horizontal Resolution | Frame Rate (Hz) | Sample Rate/ Pixel Clock (MHz) | Total Horizontal Resolution (A) | Horizontal Blanking (B) |
|---|---|---|---|---|
| 1920 | 23.976 | 74.176 | 2750 | 830 |
| 1920 | 24 | 74.25 | 2750 | 830 |
| 1920 | 25 | 74.25 | 2640 | 720 |
| 1920 | 29.97 | 74.176 | 2200 | 280 |
| 1920 | 30 | 74.25 | 2200 | 280 |
| 1440 | 23.976 | 55.632 | 2062.5 | 622.5 |
| 1440 | 24 | 55.6875 | 2062.5 | 622.5 |
| 1440 | 25 | 55.6875 | 1980 | 540 |
| 1440 | 29.97 | 55.632 | 1650 | 210 |
| 1440 | 30 | 55.6875 | 1650 | 210 |
| 1280 | 23.976 | 49.451 | 1833.3 | 553.3 |
| 1280 | 24 | 49.5 | 1833.3 | 553.3 |
| 1280 | 25 | 49.5 | 1760 | 480 |
| 1280 | 29.97 | 49.451 | 1466.7 | 186.7 |
| 1280 | 30 | 49.5 | 1466.7 | 186.7 |
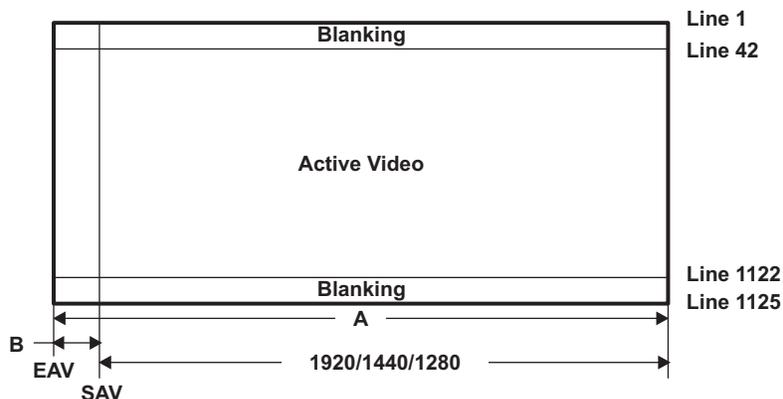
**Figure 10. SMPTE274M Progressive Vertical and Horizontal Components**
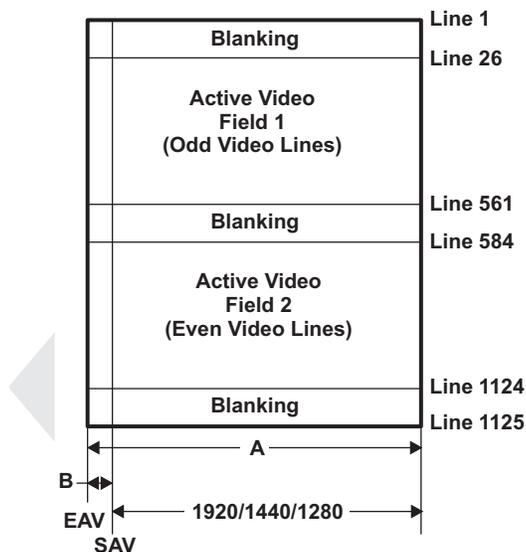
See Table 3 for A and B pixel counts. Lines 42 – 1122 make up the 1080 horizontal lines of active video. The vertical lines of active video are defined by the total horizontal resolution (A) minus the horizontal blanking count (B).

**Table 3. Various SMPTE274M Progressive Frame Rates**

| Active Horizontal Resolution | Frame Rate (Hz) | Sample Rate/ Pixel Clock (MHz) | Total Horizontal Resolution (A) | Horizontal Blanking (B) |
|---|---|---|---|---|
| 1920 | 23.976 | 74.176 | 2750 | 830 |
| 1920 | 24 | 74.25 | 2750 | 830 |
| 1920 | 25 | 74.25 | 2640 | 720 |
| 1920 | 29.97 | 74.176 | 2200 | 280 |
| 1920 | 30 | 74.25 | 2200 | 280 |
| 1920 | 50 | 148.5 | 2640 | 720 |
| 1920 | 59.94 | 148.352 | 2200 | 280 |
| 1920 | 60 | 148.5 | 2200 | 280 |
| 1440 | 23.976 | 55.632 | 2062.5 | 622.5 |
| 1440 | 24 | 55.6875 | 2062.5 | 622.5 |
| 1440 | 25 | 55.6875 | 1980 | 540 |
| 1440 | 29.97 | 55.632 | 1650 | 210 |
| 1440 | 30 | 55.6875 | 1650 | 210 |
| 1440 | 50 | 111.375 | 1980 | 540 |
| 1440 | 59.94 | 111.264 | 1650 | 210 |
| 1440 | 60 | 111.375 | 1650 | 210 |
| 1280 | 23.976 | 49.451 | 1833.3 | 553.3 |
| 1280 | 24 | 49.5 | 1833.3 | 553.3 |
| 1280 | 25 | 49.5 | 1760 | 480 |
| 1280 | 29.97 | 49.451 | 166.7 | 186.7 |
| 1280 | 30 | 49.5 | 1466.7 | 186.7 |
| 1280 | 50 | 99 | 1760 | 480 |
| 1280 | 59.94 | 98.901 | 1466.7 | 186.7 |
| 1280 | 60 | 99 | 1466.7 | 186.7 |

## 1.5 Digital LCD VGA Standards

For digital LCDs, the term VGA standard generally describes a display with a 640×480 pixel resolution with a frame rate of 60 fps and 16 bits per pixel. However, there is no standard followed for the timing between different LCDs. The signaling used in digital LCDs is very similar to that used in analog VGA signaling.

Digital LCDs have a common signal interface. Most LCDs each have 6 or 8 bits of red, green, and blue color signals. Also, they have four control signals: the pixel clock, horizontal sync, vertical sync, and data valid. These control signals have different names depending on the LCD manufacturer. The pixel clock runs at approximately 27 MHz; new color data for one pixel is available every clock. A vertical sync occurs at the beginning of each frame. The horizontal sync occurs at the beginning of each line. There is a blanking period at the beginning and end of each line often referred to as the back porch and front porch, respectively. The data valid signal denotes when there is valid data on the color lines. This goes active between the front and back porches. The timings for all of these signals relative to each other are dependent on the actual LCD used.

The following application uses a Sharp LQ64D343 LCD. For any other LCD, the timing settings must be modified.

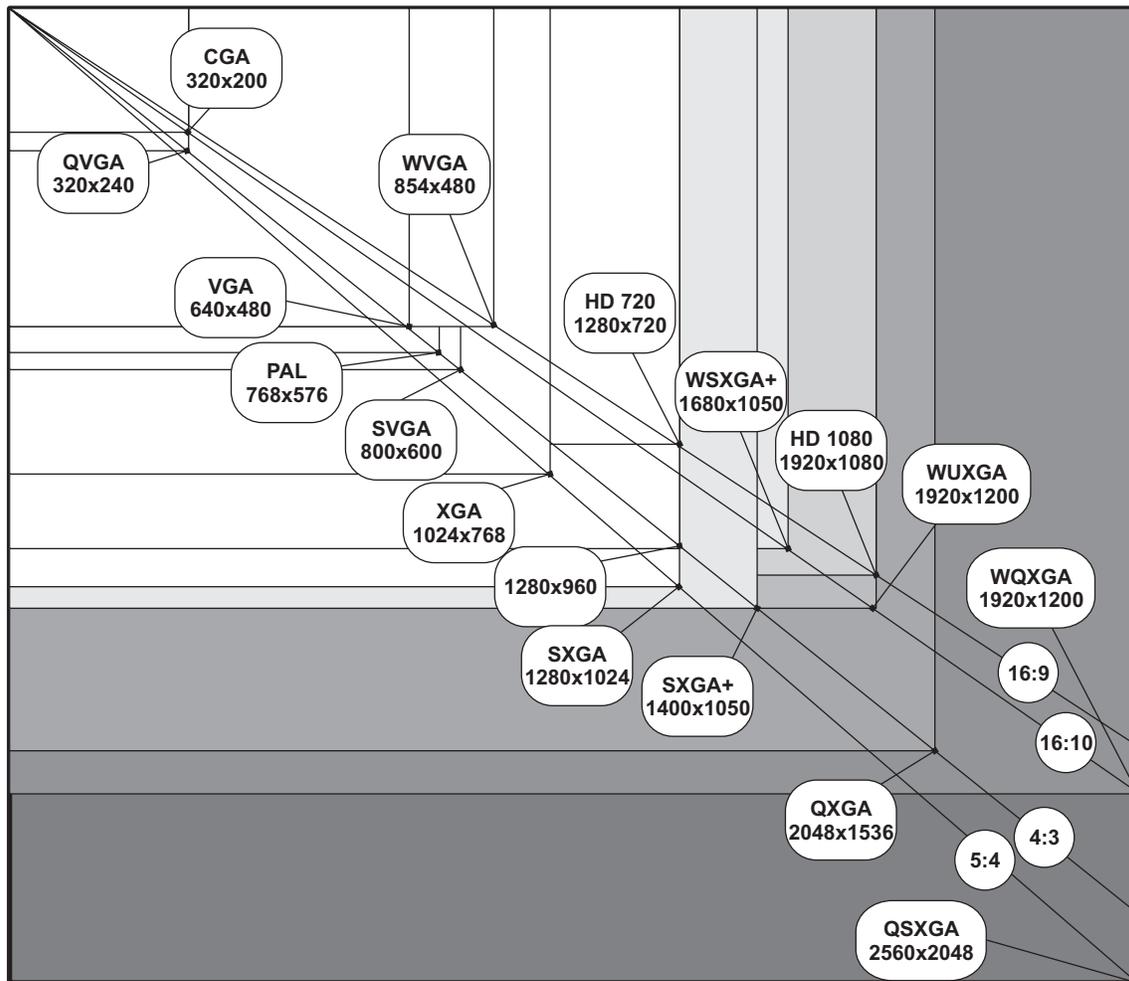Figure 11 (courtesy of Wikimedia Commons, a free media file repository) shows other standard resolutions.



**Figure 11. Common LCD Resolutions**

## 2 Hardware Overview

This section provides an overview of the hardware for each of the HD and VGA driver solutions available for the DaVinci DM6446 digital media processor.

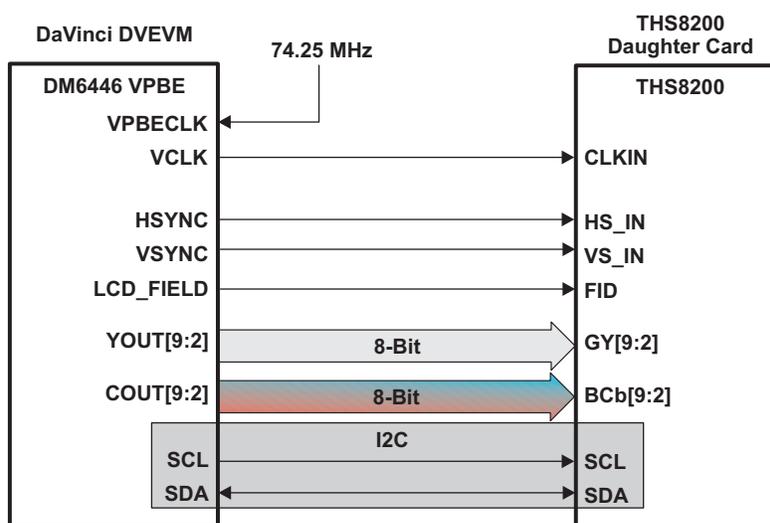### 2.1 DM6446 VPBE to THS8200 Hardware Daughter Card solution

This section describes the implementation of a high definition video using the TMS320DM6446 DaVinci device and the THS8200 video design analog converter (DAC), both available from Texas Instruments. Both the THS8200 and the TMS320DM6446 are capable of supporting different digital video pixel formats. Since the 720p and 1080i Society of Motion Picture and Television Engineers (SMPTE) standards recommend YCbCr4:2:2 pixel-format, this was the pixel format chosen for this hardware configuration. This mode is commonly referred to Y/C mode since it uses two data buses, a Y bus for Luma component and a C bus in which Cb and Cr chroma components are multiplexed.

Table 4 outlines the hardware capabilities of the TMS320DM6446 video processing back end (VPBE) and the THS8200.

**Table 4. Hardware Overview**

| Feature | DM6446 VPBE | THS8200 |
|---|---|---|
| Maximum Resolution | 4096×2048 | 8192×2048 |
| Timing | Master or Slave<br>Progressive or Interlaced | Master or Slave<br>Progressive or Interlaced |
| Pixel Formats | YCbCr4:2:2 (8 or 16-bit)<br>RGB 888, RGB 666, RGB 565 | Flexible 10, 20 or 30-bit interface to support variety of YCbCr and RGB formats (includes YCbCr 4:2:2) |
| Maximum Pixel Clock | 75 MHz | 205 MHz |

Table 4 indicates that the hardware supports the 720p and 1080i video standards. In particular, both hardware parts can support the higher 1920×1080 resolution required by 1080i, the 74.25 pixel clock required by 720p and 1080i standards, and the Y/C 4:2:2 pixel format. In addition, both hardware parts act as either a master or a slave for video timing, so there is flexibility as to which device provides the timing information. For this particular implementation, TMS320DM6446 was configured as the master and provided timing signals to the THS8200. Figure 12 shows how these two hardware components were connected. Figure 13 shows a top level block diagram of the hardware setup that demonstrates NTSC and digital output video display. In addition, there is a schematic of the THS8200 daughter card available with this application report. For more information, see the device-specific data sheets on these parts.



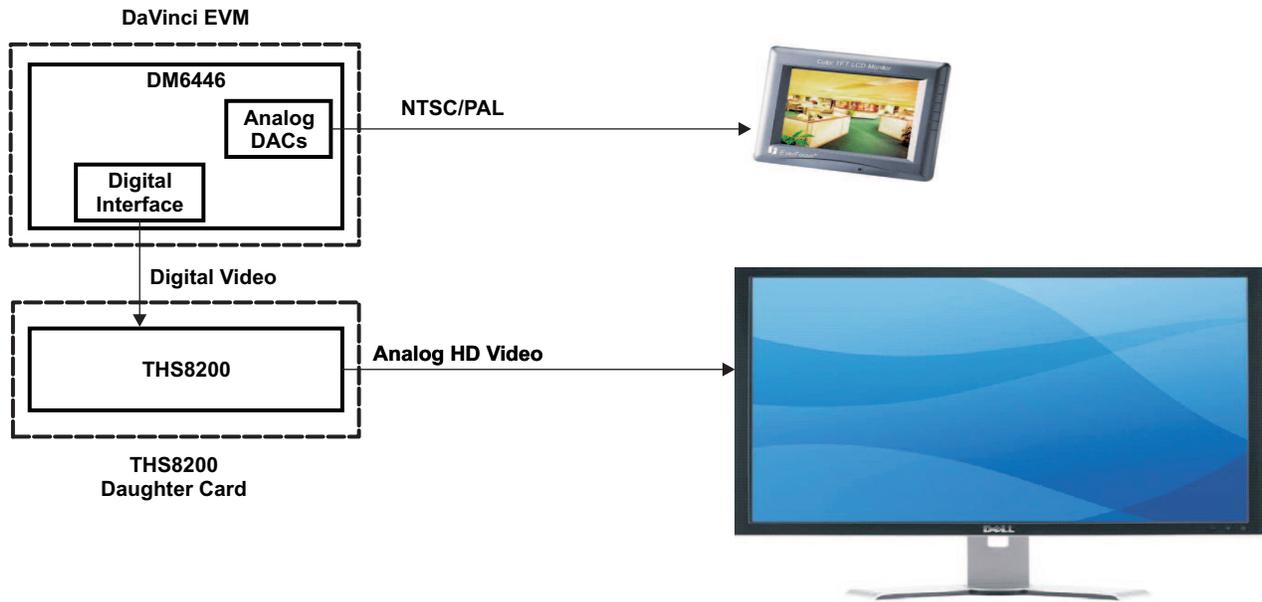**Figure 12. Hardware Connection Diagram**

**Figure 13. Top Level Block Diagram**

## 2.2 DM6446 VPBE to Logic PD Hardware Daughter Card Solution

Logic PD is a TI third party partner and provider of contract engineering services and TI CODEC software for a variety of TI parts, including the DaVinci series. To augment the capabilities of various processor development kits, including the digital video evaluation module (DVEVM), Logic PD also manufacturers and sells a line of LCD evaluation kits in most popular screen sizes. Logic's 6.4" VGA resolution display kit is used as an example in this application report, and the provided software driver contains timing parameters compatible with this display. Both the display itself and the daughter card connector for the DM6446 DVEVM can be purchased from Logic PD.

Because no video encoder is needed to interface RGB output to LCD display, the hardware is limited to the performance of the DM6446 VPBE. Table 5 summarizes the VPBE capabilities.

**Table 5. Hardware Overview for Logic PD LCD Solution**

| Feature | DM6446 VPBE |
|---------|-------------|
| Maximum Resolution | 4096 × 2048 |
| Timing | 4096 x 2048 Progressive or Interlaced |
| Pixel Formats | YCbCr4:2:2 (8 or 16 bit) RGB 888, RGB 666, RGB 565 |
| Maximum Pixel Clock | 75 MHz |

Table 5 indicates that the hardware supports the VGA video standards. TMS320DM6446 was configured as the master and provided timing signals to the display device. Figure 14 shows how DM6446 and Logic PD components were connected. Figure 15 shows a top level block diagram of the hardware setup that demonstrates NTSC and RGB output LCD display. In addition, there is a schematic of the Logic PD daughter card available with this application report. For more information, see the device-specific data sheets on these parts.
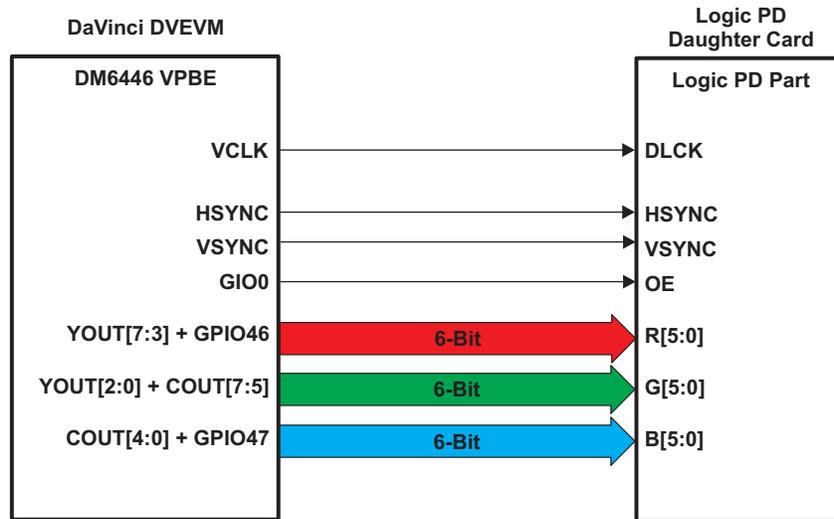


**Figure 14. Hardware Connection Diagram**

Level shifters are required on all the LCD signals from the DVEVM because the LCD requires logic levels different than 1.8 V. This LCD requires all signals to be at a logic level of 3.3 V, so it was necessary to translate the 1.8 V logic signals to 3.3 V logic signals.
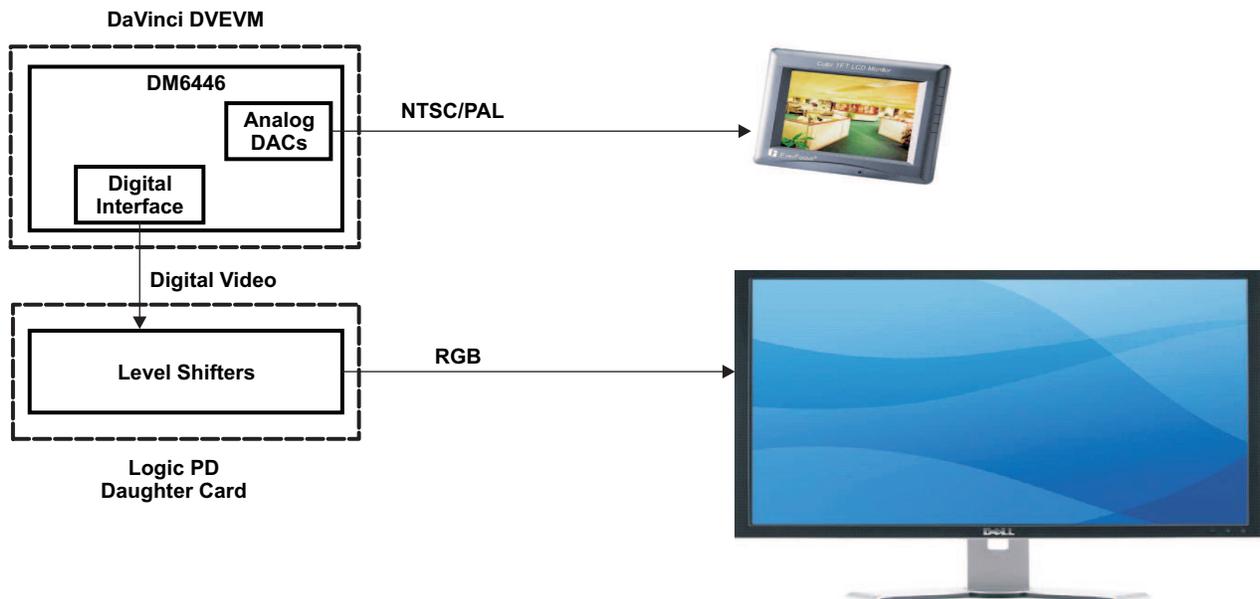


**Figure 15. Block Diagram of Interface to LCD**

### 2.2.1 Logic PD LCD Display Availability

Logic's display kits are available through the most electronics distributors. See www.logicpd.com for a current listing of sales partners.

The adapter required to connect the 6.4" VGA display kit to the DVEVM is available directly from Logic. Send an email to dsp_team@logicpd.com for more information on this adapter.

## 3 Software Overview

This section provides a brief overview of the software. The DaVinci evaluation module (EVM) software architecture is broken down into three main areas: an application layer, a signal processing layer, and a Linux® driver layer, as shown in Figure 16.
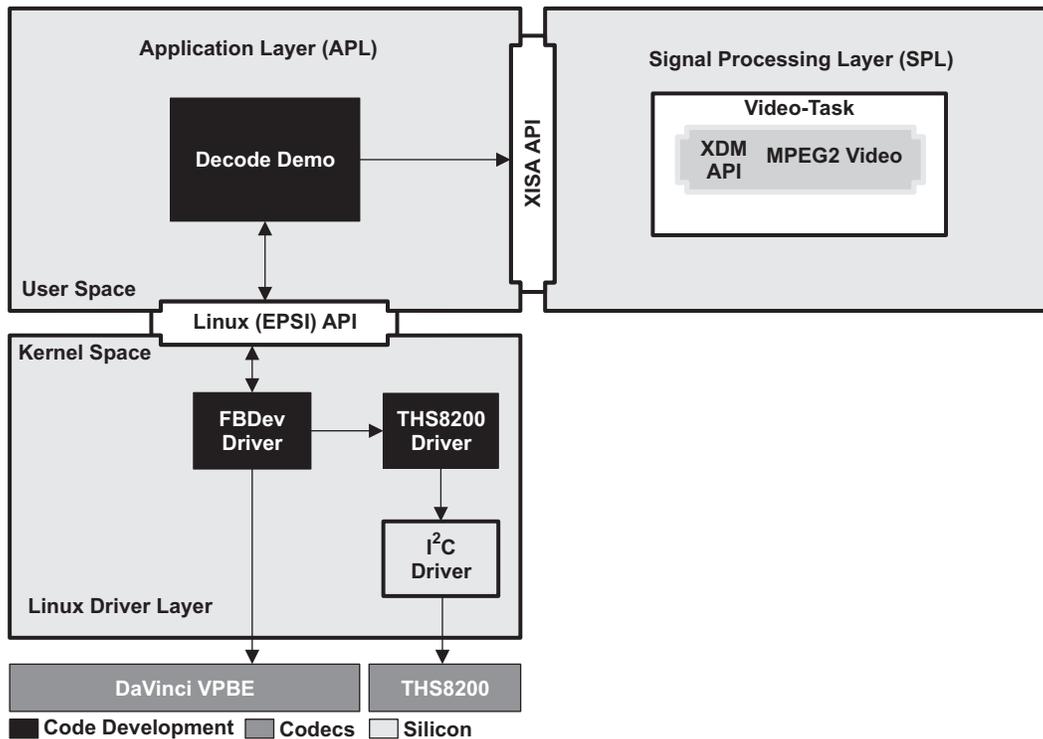


**Figure 16. DVEVM Software Architecture Overview**

The software components that were developed or modified to implement the HD and VGA solutions are shaded in black in Figure 16.

### 3.1 *Linux Frame Buffer Driver (FBDev) Overview*

In Linux, character drivers are classified as streams that access data in a first-in-first-out (FIFO) order, similar to accessing files. The standardized frame buffer driver (FBDev) fits into this category. There are five system calls normally supported by character devices such as FBDev, these are:

**open (char * pathname, int flags)—**Opens or creates the device specified by *pathname*; the *flags* parameter can specify if the device is opened for read, write, or both. If successful, this function returns a handle used for all future accesses to this device.

**close(int fd)—**Closes the device specified by the *fd* handle

**read (int fd, void\* buf, size_t count) —**Reads a maximum of *count* bytes from the device associated by *fd* into buffer *buf*.

**write(int fd, void\* buf, size_t count)—**Writes *count* bytes from buffer *buf* to device referenced by *fd*.

**ioctl (int fd, int request, …)**—Performs a device *request* on the specified *fd* device; the third argument is normally a pointer to a request specific data structure.

In addition, there are two secondary calls supported by a large number of character devices, these are:

**mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)**—Maps *length* bytes starting at *offset* from the file *fd*, preferably beginning at *start*; *prot* and *flags* to comparably describe access and sharing properties (use *man mmap* Linux command for more details).

**munmap(void *start, size_t length)**—Deletes the mapping beginning at start.

Table 6 lists the standard requests (also know as IOCTLs) specific to all FBDev drivers.

**Table 6. IOTCL Requests Defined in FBDev Standard**

| IOCTL | Description |
|---|---|
| FBIOGET_VSCREENINFO | Copies the current fb_var_screeninfo structure to you |
| FBIOPUT_VSCREENINFO | Called when you pass a structure to set the parameters such as:<br>• X and Y resolution<br>• Margins on the top, bottom, left and right<br>• Bits per pixel mode and size and offset of color components in 1 pixel data |
| FBIOGET_FSCREENINFO | Returns the *fixed* information about the frame buffer such as the start and length of frame buffer memory, length of a line, flags for hardware accelerations, if any |
| FBIOPAN_DISPLAY | To view a particular portion of the frame buffer memory<br>Can be used for double/triple buffering by allocating virtual frame buffer size twice/three times the required (screen) size and viewing two halves alternately |
| FBIO_WAITFORVSYNC | Call made by applications to block until the next vsync<br>• Assures that a frame has been displayed |
| FBIO_WAITFORVSYNC | Ioctl for setting the attribute blending levels |
| FBIO_SETPOSX | To set the X position of the window |
| FBIO_SETPOSY | To set the Y position of the window |

Since all FBDev drivers must support these common sets of IOCTLs requests (not to mention standard character device system calls), there is an opportunity for code reusability. As shown in Figure 17, much of the FBDev driver is supported by standard Linux kernel modules, taking the burden off of the device manufacturer(s) by allowing them to focus on the hardware specifics.
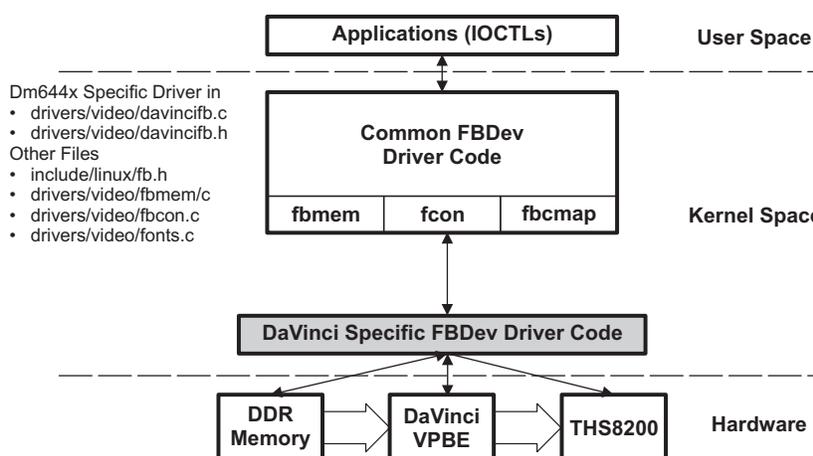


**Figure 17. FBDev Software Driver Architecture**

## 3.2 DaVinci EVM FBDev Overview

This section examines the FBDev driver as implemented for the DaVinci EVM, followed by a block diagram showing the flow of calls made to these functions. Table 7 lists the the most important driver functions, with a brief description of each.

**Table 7. Functions Defined in DaVinci FBDEV Driver**

| Function | Description |
|---|---|
| **Standard Entry Points Defined for Linux Drivers** | |
| davincifb_init | Registers device and driver; note that the secondary entry points are defined as part of the driver registration process |
| davincifb_cleanup | Unregisters device and driver |
| **Standard Secondary Entry Points for Linux Drivers** | |
| davincifb_probe | Entry point called during device initialization to probe its capabilities. This is where the driver registers its ISR, sets default background color, and initializes the DM6446 hardware for the default National Television System Committee (NTSC) output. |
| davincifb_remove | Called by the system for device to free resources and prepare for removal. |
| **Standard FBDev Entry Points** | |
| davincifb_check_var | Checks to ensure the settings being requested can be satisfied by the hardware |
| davincifb_set_par | Called to configure hardware for new output format |
| davincifb_setcolreg | Sets a color register (optional: used to program CLUT) |
| davincifb_blank | Blanks the display |
| davincifb_pan_display | Pans the display |
| davincifb_ioctl | Handles any IOCTLs not handled by upper layers |
| init_fb_info | Standard Linux application programming interface (API) used to register new frame buffer device. As part of the registration information, we pass the capabilities of the device as well as the FBDev entry points. |
| **Helper Functions** | |
| davinci_isr | Interrupt service routine that gets VSYNC events |
| davincifb_ntsc_composite_config | Configures hardware for NTSC composite video output |

Although only two helper functions were discussed in Table 7, there are actually many helper functions (in italics text) as shown in Figure 18. Most of the details of the helper functions are irrelevant to showing how HD and VGA support are added; the helper functions not mentioned in the table are small, self explanatory, and unchanged. The same color-coding scheme in Table 7 applies to Figure 18.
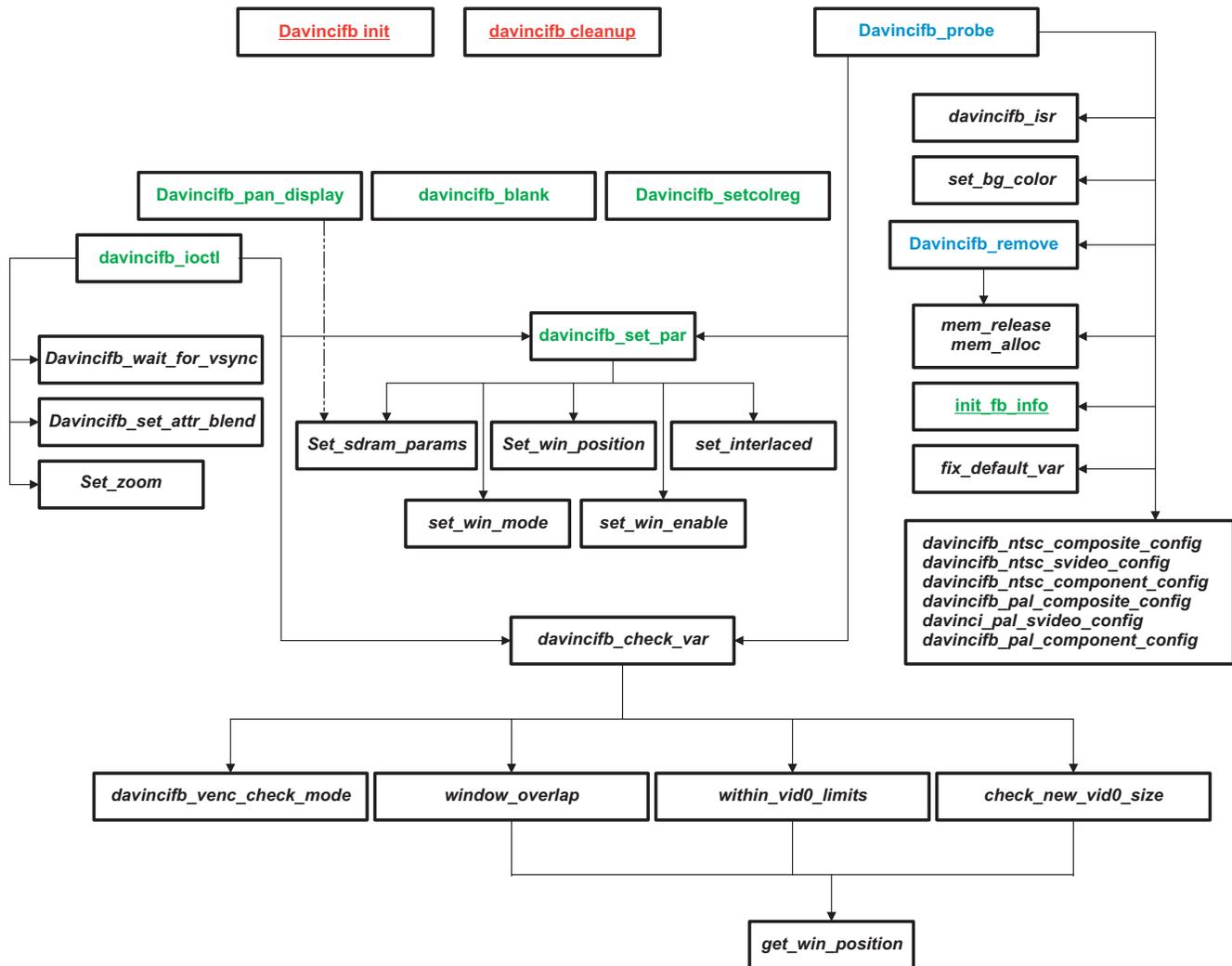


**Figure 18. DaVinci FBDev Software Driver Architecture**

The hardware initialization takes place in the davincifb_probe. During this phase the driver calls davinci_ntsc_composite_config to configure TMS320DM6446 to output NTSC composite video. Note that the video output format (NTSC or PAL) is determined by a u-boot environment variable at boot time. For more information, see the *DVEVM Getting Started Guide* (SPRUE66). In this driver model, video output determination happens once during driver initialization (as part of the boot-up sequence) and you cannot change this video output format after the system starts. This limitation has been removed when adding hardware and VGA support.

## 3.3 DaVInci EVM Decode Demo Overview

Similar to the driver, the decode demo that ships with the DVEVM assumes one constant configuration defined at boot time (via the u-boot environment variable). Therefore, there is no need to provide it in the information when running the decode demo from the command line (for instance, /decode -l -v data/video/davinvi_effect.m2v). The demo source code supplied with this application report has been modified to provide a video resolution via the –r option, allowing run-time switching among the standard definition (SD), high definition (HD), and video graphics adaptor (VGA) video output modes supported by the provided software. Figure 19 shows a timing diagram showing the flows of requests from the original decode demo to the original FBDev driver.
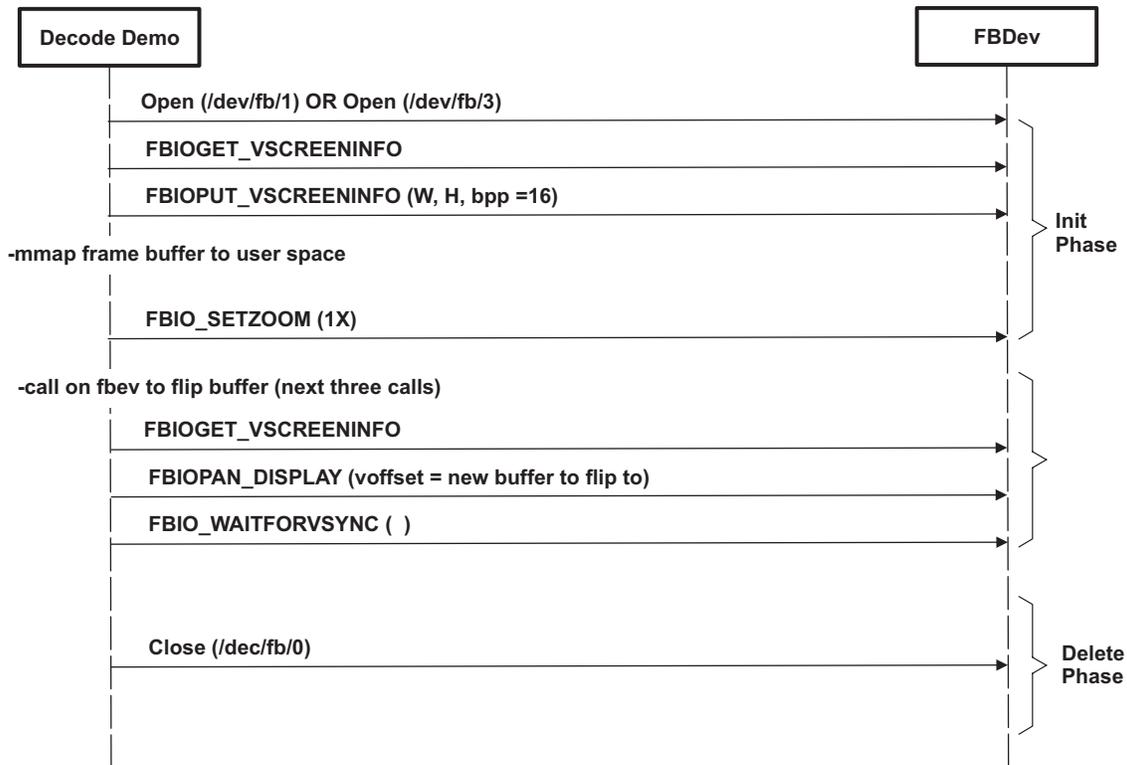


**Figure 19. Timing Diagram Showing Decode Demo and FDBDev Interaction**

Many of the system calls and IOCTL requests communicate with the driver. The upper layers of the drivers do as much as they can and forward what they cannot do to the hardware-specific layer of the driver (davincifb.c in this case).

## 3.4 FBDev Driver Modifications for HD and VGA Support

This section provides an overview of how HD and VGA support were added to the source code. As previously discussed, the -r flag has been added to the decode demo which allows you to specify the output format you want to display (for instance, ./decode -I -r 720p -v data/video/city_720p.m2v). The decode demo timing diagram stays the same. The FBIOPUT_VSCREENINFO IOCTL request contains the necessary information for the output format that is displayed; this IOCTL eventually translates to the driver call function davincifb_set_par. This function must be changed so it takes the responsibility of configuring the hardware for the desired output format (instead of this being done once during driver initialization). Figure 20 shows the new block diagram for the driver. Many of the details associated with other blocks have been removed for clarity because davincifb_set_par and davincifb_probe were the only functions that changed.
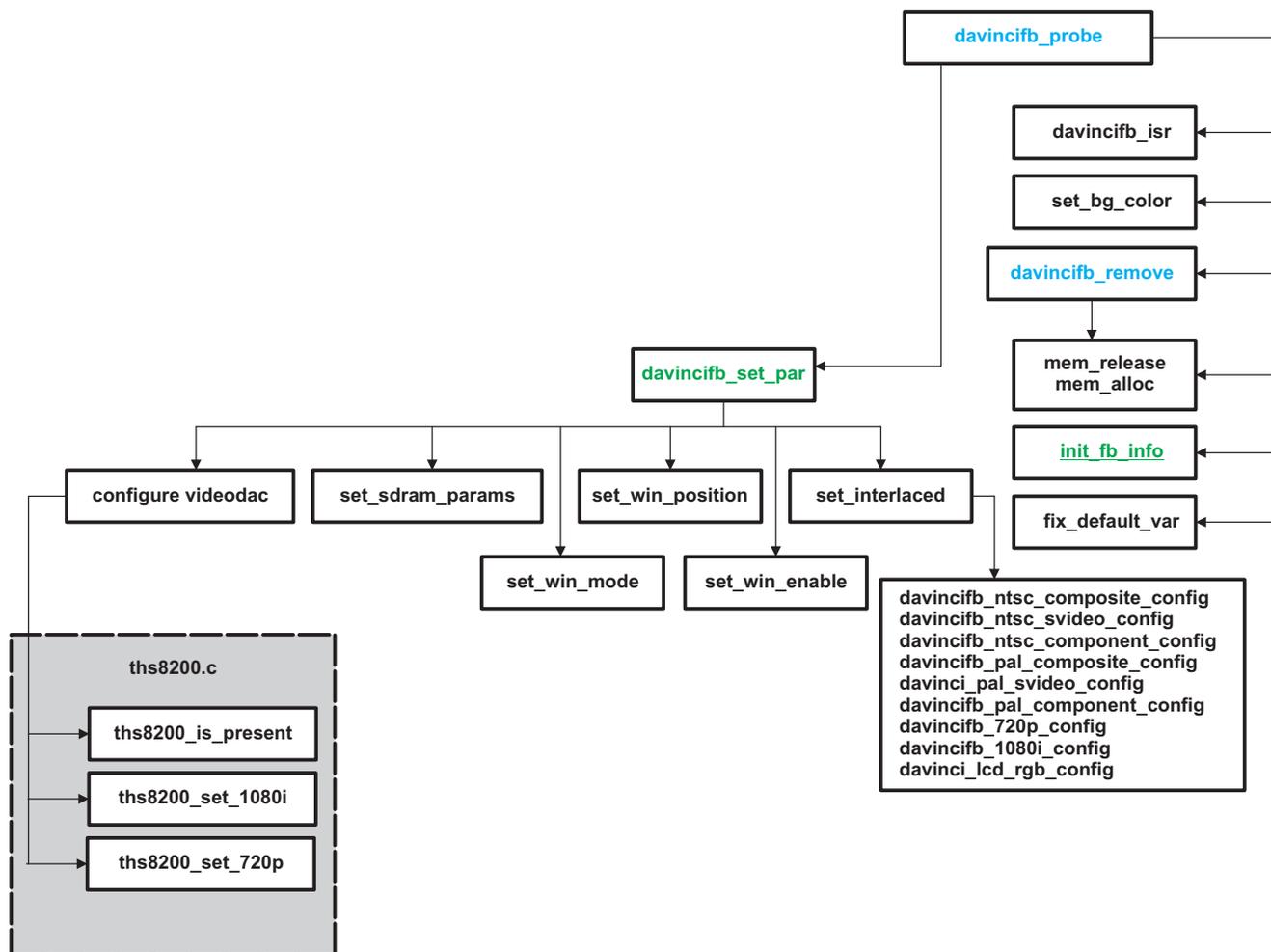


**Figure 20. Changes to DaVInci FBDev Driver to Support HD/VGA Resolutions**

The davincifb_set_par now calls on the various helper functions to configure the TMS320DM6446 hardware for various output formats, including 720p, 1080i and VGA. A configureVideoDAC function was added that calls on functions defined in another source file to configure the THS8200 for the desired configuration. This action keeps the source code as modular as possible, so that the THS8200 part can be replaced with another DAC part with very little effort.

The remainder of this section describes the changes made to the official release version of the DaVinci FBDev Linux driver that ships with the DVEVM, as well as the challenges encountered when adding digital output support.

### 3.4.1 Original DVEVM Release Options (NTSC/PAL)

The original driver was hardcoded to work with NTSC and PAL video formats only; therefore, much of the code contained pre-defined buffer sizes, window sizes and register settings were set once during initialization and never changed again. One of the first challenges encountered was to study the driver architecture to determine how to add digital output support without changing the existing behavior (e.g., NTSC by default) so the driver continued to be backward compatible with existing demos. To overcome this, we

- Needed to allocate buffers large enough to accommodate the largest supported video format (1080i) at initialization time. This was done at the application (decode demo) and driver layers by adding new *#defines* for the newer resolutions being supported and adjusting max buffer sizes accordingly.

- Needed the decision making process of how the registers were configured moved out of the initialization phase and onto a place accessible by applications at run-time. It was discovered that applications need to make FBIOPUT_VSCREENINFO IOCTL calls to the driver to initialize parameters such as video window size; this call from the application to the driver results in a call to the driver's fb_set_par entry point which all FBDev drivers are required to define. At run-time, this was the ideal place to see what the application requested and decide which video format to use to configure the TMS320DM6446. Optionally, a new proprietary IOCTL could have been created for accomplishing this, but the standard FBIOPUT_VSCREENINFO IOCTL was chosen for backward compatibility and maximum portability in the future.

- New driver functions were created, adding additional support whenever possible rather than changing existing functions, ensuring backward compatibility. In this step, we created one supporting function which switches between analog DAC output and digital output on the TMS320DM6446 (to be used by all video formats that this functionality) as well as a supporting function for each new video resolution for setting up the TMS320DM6446. Finally we added functions to call the THS8200 driver to set it up correctly.

- Added support to decode demo application to allow video format parameter to be entered from the command line.

### 3.4.2 720p/1080i Additions With THS8200

To support 720p and 1080i, we added the following support functions to the driver. The actual register configurations made in these functions are discussed in more detail in **Appendix A.3** and **Appendix A.4**.

**void enableDigitalOutput(int bEnable): —**Configures the TMS320DM6446 to switch between analog and digital mode. This is used by all video formats that need this functionality.

**void davincifb_720p_config(int on) —**Configures the TMS320DM6446 for 1080ip video format.

**ths8200_set_720P(void) —**Configures the THS8200 for 720p video format.

**ths8200_set_1080I(void) —**Configures the THS8200 for 1080i video format.

The THS8200 related functions are defined in a separate source file (ths8200.c) that can be easily replaced by similar source code specific to another part; the details of doing this are discussed in a later section.

In addition to the above changes, support for a -r option was added to the decode demo application that allows you to specify the video resolution of the media file to be decoded from the command line. For example, to decode a 720P file, type something similar to the following on the command line:

./decode_hd -l -r 720p -v <location of 720p video stream video file>

### 3.4.3 LCD RGB 640×480 VGA Addition

To integrate support for the Logic PD LCD display to the davincifb driver, a few additions and modifications must be made.

First, add the function davincifb_lcd_rgb_config to set up the registers of the VENC to output red, green, blue (RGB) data, and control signals. This function configures the video processing back end (VPBE) registers to output RGB666 non-interlaced color data and control signals. In addition, this function configures the timings for the LCD by calling another function (set_lcd_timings) that was added. This function call sets up all timings required to make the LCD work properly. This function is found in the source code of the driver patch that comes with this application report. To ease the use of this function, a structure called lcd_timing_params was created with all of the LCD timing data in it. This structure contains all of the timing parameters that must be set to ensure proper timing on the LCD.

### 3.4.4 Run-Time Resolution Switching

With the source code included with this application report, you can specify the video resolution dynamically from the command line. This is a big improvement over the static resolution available on the current 1.10 software release. This section discusses this process in more detail.

The encode demo application takes the video format input by the user via the -r option, sets up the appropriate application level buffer sizes, and makes a FBIOPUT_VSCREENINFO IOCTL call to the driver. The driver processes this request in its fb_set_par entry point. In this driver, this is in a function called davinci_set_par; consequently, the driver analyzes the requested window size, determines the corresponding video format, and configures the TMS320DM6446 and THS8200 registers.

### 3.4.5 Video Encoder IOCTL

To develop code that is as modular as possible, proprietary IOCTL was added called FBIO_SET_VIDEODAC to allow you to select one of the various digital DAC parts connected to DaVinci's TMS320DM6446 VPBE. Upon initialization, the driver defaults to the DAC_AUTO_DETECT mode, where the driver checks for the presence of a list of DACs until it finds the first one and makes this the default DAC. At present, the THS8200 is the only DAC supported in the list. Once the driver detects the presence of the THS8200, it makes it the default DAC. Future versions of the driver will allow you to add support for another DAC part, add it to the list, and dynamically select (via FBIO_SET_VIDEODAC IOCTL) which part you want to use from the application. Generic wrapper functions were also added for setting up different video resolutions on the DAC side (for instance., THS8200 side); this ensures that any changes related to DAC parts occur in one place as opposed to many.

### 3.4.6 Adding Custom Video Encoder Capabilities/Additional Resolutions

To add support for a new DAC part, you must do the following:

1. Add a new enumeration value for DAC in the davincifb.h file.
2. Write supporting source code similar to the ths8200.c file that exports two APIs at minimum (in the THS8200 case, 3 new APIs were exported), one the driver calls to determine the presence of the DAC and another for each video resolution the DAC supports.
3. Modify void configureVideoDAC(int videoFormat) function in the davincifb.c, as this is the only function that should ever call into the functions exported in step 2.

To add support for additional resolutions, you must:

1. Create a new static function (only accessible by the corresponding source file) in the davincifb.c file that sets up all the TMS320DM6446 registers for your new video resolution.
2. Create a new exported function in the ths8200.c file (or similar, if using a different DAC) to configure the ths8200 (or different DAC) registers for your new video resolution.
3. Add source code logic to identify your new resolution (for instance, window size) in the davinci_set_par function found in the davincifb.c file. Once the code segment is added properly and identifies an application request for your new resolution, have it call the functions created in steps 1 and 2. Please note that DAC function is called via the generic configureVideoDAC wrapper function.

### 3.4.7 Color Space Conversion

Internally, DaVinci handles all of its video in YCbCr422 format. When the VPBE is set up for RGB output, the DaVinci VPBE automatically converts the color data from YCbCr space to RGB space. DaVinci supports RGB output with a precision of up to 24 bits per pixel. The process of configuring the color space converter of the VENC is discussed later.
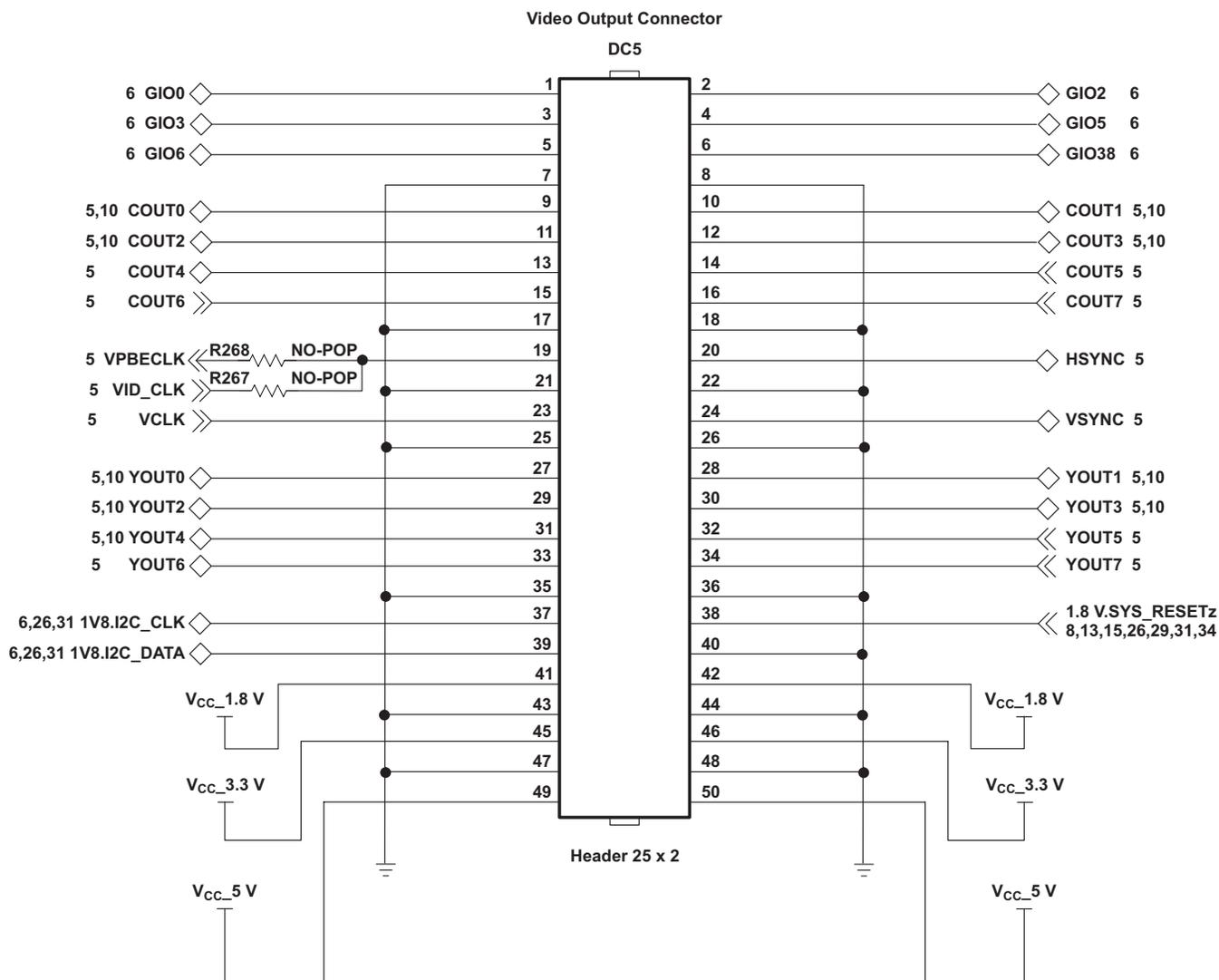
## 4 Installing the FBDev Driver Patch and Running the Decode_HD Demo

This section demonstrates the process for installing the FBDev driver patch for HD and LCD support, rebuilding the Linux kernel, building the decode_hd application, and exercising the new driver capabilities on a DM6446 DVEVM with the required daughter card hardware.

### 4.1 DM6446 DVEVM VPBECLK Modification

By default, the DM6446 DVEVM does not connect the VPBECLK signal to the video out daughter card connector (DC5). To drive a video display pixel clock at a frequency other than 27 MHz, the VPBECLK signal must be connected to an external oscillator. For the 720p and 1080i configurations, a 74.25 MHz pixel clock is required; therefore, the DM6446 DVEVM must be modified to connect the video out daughter card connector to the VPBECLK of the DM6446.

Fortunately, this can be done by populating a 0 Ω resistor across resistor pads R268 on the bottom side of the DM6446 DVEVM. Figure 21 shows the DM6446 DVEVM schematic where this resistor must be populated.

**Video Output Connector**

**DC5**



**Figure 21. DM6446 DVEVM Schematic: Video Out Daughter Card Connector**

## 4.2 Installing the FBDev Driver Patch

Along with this application report is a zip file containing a driver patch with the new 720p, 1080i, and digital LCD display support. The driver patch contains six source files that must be patched into the DM6446 LSP source tree. Currently, this driver patch has been tested against DM6446 LSP v1.20. For this reason, developers should upgrade their DM6446 DVEVM LSP to v1.20 before proceeding with the driver installation.

To install the driver:

1. Download and install the mvl_lsp_setuplinux_1_20_xx_xx.bin file from the DaVinci Software Update Site located at https://www-a.ti.com/extranet/cm/product/dvevmsw/dspswext/general/homepage.shtml. This is required for applying the updated driver.

2. Download and save the attached source code that comes with this application report. Inside the lsp_1_20 directory, there are six files of the LSP that must be updated to take advantage of the latest driver updates.

3. Back up the FBDev driver source files.

    a. cd …/lsp/ti-davinci/drivers/video

    b. cp davincifb.c davincifb.c.1.20

    c. cp Makefile Makefile.1.20

4. Install the new FBDev driver source files.

    a. cd …/lsp/ti-davinci/drivers/video

    b. cp .../lsp_1_20/davincifb.c

    c. cp .../lsp_1_20/Makefile

    d. cp …/lsp_1_20/ths8200.c

5. Back up and install the new FBDev driver header file.

    a. cd …/lsp/ti-davinci/include/video

    b. cp davincifb.h davincifb.h.1.20

    c. cp .../lsp_1_20/davincifb.h

6. Back up and install the new memory.h header file.

    a. cd …/lsp/ti-davinci/include/asm-arm/arch

    b. cp memory.h memory.h.1.20

    c. cp .../lsp_1_20/memory.h

7. Back up and install the new mmzone.h header file.

    a. cd …/lsp/ti-davinci/include/linux

    b. cp mmzone.h mmzone.h.1.20

    c. cp .../lsp_1_20/mmzone.h

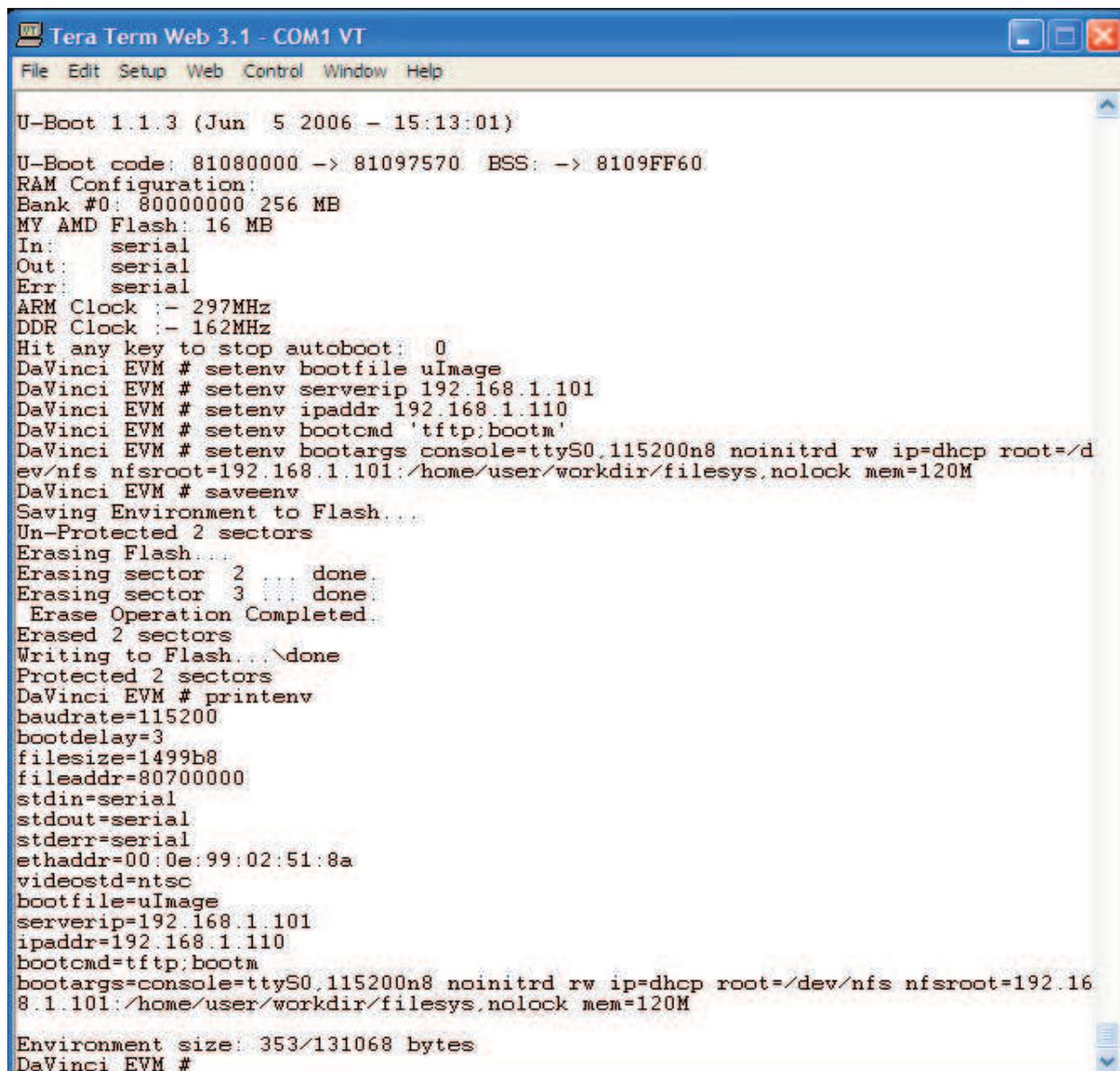## 4.3 Rebuilding Linux Kernel With New LSP Drivers

Now that the new drivers have been installed, the LSP must be re-built to generate a new uImage Linux kernel that can be booted either via the TFTP server or flashed into the external inverted OR gate not OR (NOR) or inverted AND gate not AND (NAND) flash memory of the DM6446 DVEVM. To complete this section, the build tools (mvl_setuplinux_1_20_xx_xx.bin), target file system (mvl_target_setuplinux_1_20_xx_xx.bin), and LSP (mvl_lsp_setuplinux_1_20_xx_xx.bin) must all be installed on the host workstation. For additional details, see the *DVEVM Getting Started Guide* (SPRUE66) document.

To rebuild the LSP:

1. Go to the main directory where the updated LSP is installed.

    a. cd …/lsp/ti-davinci

2. Set the standard DaVinci DM6446 DVEVM configuration.

    a. make ARCH=arm CROSS_COMPILE=arm_v5t_le- davinci_dm644x_defconfig

3. Rebuild the Linux LSP and generate the uImage Linux Kernel.

    a. make ARCH=arm CROSS_COMPILE=arm_v5t_le- uImage

4. Copy the Linux Kernel to the TFTP Boot directory.

    a. cp …/lsp/ti-davinci/arch/arm/boot/uImage /tftpboot/uImage

5. Restart the DM6446 DVEVM and press any key to enter U-Boot.

6.  Configure U-Boot for the TFTP booting Linux Kernel and the NFS File System using the steps below. See Figure 22 and the *DVEVM Getting Started Guide* (SPRUE66) for more information.

    a.  setenv bootfile uImage
    b.  setenv serverip <IP address of host machine>
    c.  setenv ipaddr <IP address of DVEVM>
    d.  setenv bootcmd tftp;bootm
    e.  setenv bootargs console=ttyS0,115200n8 noinitrd ip=dhcp rw root=/dev/nfs nfsroot=<Host IP>:<File System Path>,nolock mem=120M
    f.  saveenv
    g.  boot



**Figure 22. DM6446 U-Boot Configuration**

## 4.4 Building and Running the decode_hd Application

The decode_hd application demonstrates each of the new 720p, 1080i and VGA modes, taking advantage of the new Linux FBDev drivers that have been installed. This application is provided in the attached code under the decode_hd directory. For additional information, see the *DVEVM Getting Started Guide* (SPRUE66).

To build and run the decode_hd application:

1. Download and install the DM6446 DVEVM demo applications (dvevm_setuplinux_1_20_xx_xx.bin) from the DaVinci Software Update Site: https://www-a.ti.com/extranet/cm/product/dvevmsw/dspswext/general/homepage.shtml.
2. Go to the DVEVM demos directory and copy in the decode_hd source code.
   a. cd …/dvevm_1_20/demos
   b. mkdir decode_hd
   c. cd decode_hd
   d. cp …/decode_hd/* .
3. Rebuild decode_hd demo and copy the executables to the NFS file system.
   a. cd …/dvevm_1_20/demos
   b. make
   c. cp debug/decode_hdd <NFS path>/opt/dvevm
   d. cp release/decode_hd <NFS path>/opt/dvevm
4. Copy 720p and 1080i video content to the NFS file system.
   a. cp …/video_content/* <NFS path>/opt/dvevm/data/videos/*
5. Boot DM6446 DVEVM with the new LSP drivers and NFS root file system.
6. Run the decode_hd application with each of the new driver modes on a DM6446 DVEVM. Note that *decode_hd –h* provides a listing of all the decode_hd command line parameters that are available.
   a. cd /opt/dvevm
   b. ./loadmodules.sh
   c. ./decode_hd –l –r 720p –v data/videos/city_720p.m2v
   d. ./decode_hd –l –r 1080i –v data/videos/atsc_1080i.m2v
   e. ./decode_hd –l –r vga –v data/videos/davincieffect_ntsc.m2v

## Appendix A  Schematics and Software

### A.1  THS8200 Daughter Card Schematic

The schematic for this daughter card is included in the zip file which accompanies this application report.

### A.2  Logic PD LCD Daughter Card Schematic

The schematic for this daughter card is included in the zip file which accompanies this application report.

### A.3  720p Firmware Configurations

This section discusses the registers that must be programmed on both the TMS320DM6446 and THS8200 parts to achieve the 720P mode, including why you should program these registers. It focuses on the relevant registers, instead of going through the driver code line by line.

There are three main areas that must be programmed correctly to output 720P video correctly. First, the OSD Window(s) on the TMS320DM6446 must be set appropriately per the SMPTE296M standard. Second, VPBE VENC on the DM6446 must be programmed to generate the appropriate video timing and pixel clock information for the THS8200 device to use. And third, the THS8200 device must be programmed to accept the corresponding window size and timing information that it will receive from the DM6446 part. In addition to these, there are a few other DM6446 registers that do not belong to the OSD or VENC hardware blocks that must be programmed. These include:

```
// Disable Field ID output since this is a progressive resolution
dispc_reg_merge(PINMUX0, 0, PINMUX0_LFLDEN);

// Set PINMUX0 register to enable LCD output
dispc_reg_merge(PINMUX0, PINMUX0_LOEEN, PINMUX0_LOEEN);
```

### A.3.1  Video Window Settings

This setting is primarily dictated by the SMPTE296M standard. Note that the TMS320DM6446 has four display windows, two video windows and two OSD windows; each of these four windows can be programmed independently. Therefore, when defining a new video format (such as 720P), you must update the corresponding window sizes of all the display window(s) you plan on using. This is done using the following code segment:

```
#define BASEX720P          0x50        // 80 pixels
#define BASEY720P          0x5         // 5 lines
#define DISP_XRES720P      1280        // active horizontal pixels per SMPTE296M
#define DISP_YRES720P      720         // active vertical lines per SMPTE296M

// Set new baseX and baseY
dispc_reg_out(OSD_BASEPX, BASEX720P);
dispc_reg_out(OSD_BASEPY, BASEY720P);

// Set VID0 window  origin and size
dispc_reg_out(OSD_VIDWIN0XP, 220);   //Actual offset is 220 + BASEX720P = 300 pixels
dispc_reg_out(OSD_VIDWIN0YP, 25);    // Actual offset is 25 + BASE&720P = 30 lines
dispc_reg_out(OSD_VIDWIN0XL, DISP_XRES720P);
dispc_reg_out(OSD_VIDWIN0YL, DISP_YRES720P);

// Set VID1 window  origin and size
dispc_reg_out(OSD_VIDWIN1XP, 220);
dispc_reg_out(OSD_VIDWIN1YP, 25);
dispc_reg_out(OSD_VIDWIN1XL, DISP_XRES720P);
dispc_reg_out(OSD_VIDWIN1YL, DISP_YRES720P);

// Set OSD0 window  origin and size
dispc_reg_out(OSD_OSDWIN0XP, 220);
dispc_reg_out(OSD_OSDWIN0YP, 25);
dispc_reg_out(OSD_OSDWIN0XL, DISP_XRES720P);
```

```
dispc_reg_out(OSD_OSDWIN0YL, DISP_YRES720P);

// Set OSD1 window  origin and size
dispc_reg_out(OSD_OSDWIN1XP, 220);
dispc_reg_out(OSD_OSDWIN1YP, 25);
dispc_reg_out(OSD_OSDWIN1XL, DISP_XRES720P);
dispc_reg_out(OSD_OSDWIN1YL, DISP_YRES720P);
```

SMPTE296M defines the vertical blanking on top of the active video as lines 1 through 26. The offset per the comments in the code segment above is actually 30 lines; it is often necessary to adjust this number slightly to avoid seeing invalid data at the beginning of the active video window. SMPTE296M also defines the total line length of a 720P format as 1650 pixels, while the active video is only 1280; this leaves 370 horizontal blanking pixels that must be split between the left and right side of the 1280 active video pixels. SMPTE296M is flexible in how the horizontal blanking pixels are split. This choice was made to split them as 300 on the left side (leaving 70 on the right side) of the active video. Note that the choice made here must be reflected on the THS8200 side as seen in later sections.

Finally, the windows must be enabled; for this application, the choice was made to display the 720P video in video window 0 due to the buffer bandwidth limitations associated with video window 1. Note that the OSD field inversion bit is normally set when using video window 1 as a workaround for the silicon bug described in the *TMS320DM6446 Digital Media System-on-Chip Silicon Errata* (SPRZ241). This is not necessary in video window 0, therefore this bit will be cleared. The following code segment enables video window 0 and clears the field inversion bit.

```
Enable Video Window 0 / disable video window 1
dispc_reg_out(OSD_VIDWINMD, OSD_VIDWINMD_ACT0);

// Clear OSD Field Inversion for VID0 Use
dispc_reg_out(OSD_MODE, 0);
```

### A.3.2    VPBE VENC Register Configurations

This section details how to program the VENC so that it defines the appropriate timing for the video format you want to output. This strictly follows the SMPTE296M video standard and the horizontal blanking interval used when defining the window sizes.

```
// Set Timing parameters for 720P frame (must match what THS8200 expects)
dispc_reg_out(VENC_HSPLS, BASEX720P);
dispc_reg_out(VENC_VSPLS, BASEY720P);
dispc_reg_out(VENC_HINT, 1649);
dispc_reg_out(VENC_HSTART, 300);
dispc_reg_out(VENC_HVALID, DISP_XRES720P);
dispc_reg_out(VENC_VINT, 749);
dispc_reg_out(VENC_VSTART, 26);
dispc_reg_out(VENC_VVALID, DISP_YRES720P);
dispc_reg_out(VENC_HSDLY, 0);
dispc_reg_out(VENC_VSDLY, 0);
dispc_reg_out(VENC_YCCCTL, 0);
dispc_reg_out(VENC_VSTARTA, 0);
```

The timing defined above generates the horizontal and vertical syncs output to the THS8200. Another signal that the TMS320DM6446 outputs to the THS8200 is the pixel clock (VCLK shown in Figure 12). An external clock was needed to accommodate 720P, which requires a pixel clock of 74.25 per SMPTE496M. Per the DVEVM design, the DM6446 has only a 27 MHz clock that can be used by the VENC internally; this is sufficient for NTSC video format, but not for 720P. The clock output to the THS8200, VLCK, is dependent on the external clock input to the DM6446 via the VPBECLK pin and the DCLK register settings. The DLCK registers on the TMS320DM6446 hardware component can be configured to use the clock input into VPBECLK pin and generate a clock pattern up to 64 cycles long; however, for the purposes of the solution being presented in this application report, the DLCK registers are configured to generate a clock equal to the input clock at the VPBECLK pin. The clock generated via the DLCK registers is output via the VCLK pin on the TMS320DM6446 and is used to drive the THS8200. The register settings are as follows:

```
// Enable DCLOCK
dispc_reg_out(VENC_DCLKCTL, VENC_DCKCTL_DCKEC);
```

```
// Set DCLOCK pattern
dispc_reg_out(VENC_DCLKPTN0, 1);
dispc_reg_out(VENC_DCLKPTN1, 0);
dispc_reg_out(VENC_DCLKPTN2, 0);
dispc_reg_out(VENC_DCLKPTN3, 0);
dispc_reg_out(VENC_DCLKPTN0A, 2);
dispc_reg_out(VENC_DCLKPTN1A, 0);
dispc_reg_out(VENC_DCLKPTN2A, 0);
dispc_reg_out(VENC_DCLKPTN3A, 0);
dispc_reg_out(VENC_DCLKHS, 0);
dispc_reg_out(VENC_DCLKHSA, 1);
dispc_reg_out(VENC_DCLKHR, 0);
dispc_reg_out(VENC_DCLKVS, 0);
dispc_reg_out(VENC_DCLKVR, 0);
```

The following code enables several processes:

- Enables the digital LCD output
- Enables output of horizontal and vertical syncs
- Programs the VENC to use the external VPBE CLK and enables the VENC to start outputting video and timing signals to be fed to THS8200

These steps are shown in the code segment below:

```
// Enable LCD output control
dispc_reg_out(VENC_LCDOUT, 0x1);

// turning on horizontal and vertical syncs
dispc_reg_out(VENC_SYNCCTL, (VENC_SYNCCTL_SYEV|VENC_SYNCCTL_SYEH) );

// Enable external VPBE CLK and Enable VENC
dispc_reg_out(VPSS_CLKCTL, 0x0a);

// Enable all VENC, non-standard timing mode, master timing, HD, progressive
dispc_reg_out(VENC_VMOD, (VMOD_VENC|VMOD_VMD | VMOD_HDMD) );
```

### A.3.3 THS8200 Register Configurations

The details of the THS8200 are documented in the code segment below. The register-field definitions are not covered in this document; for details of the register fields for each individual register see the *THS8200 All-Format Oversampled Component Video/PC Graphics D/A System With Three 11-Bit DACs, CGMS Data Insertion Data Manual* (SLES032). Also, note that buf[0] contains the register address and buf[1] the register data, both of which are sent across the I2C bus.

The following registers set up the input and output modes.

```
// Sets YPbPr output mode, and sets polarity of the horizontal and vertical sync signals
buf[0] = 0x82;
buf[1] = 0x1b;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x82\n");

// Accept 20-bit YCbCr data
buf[0] = 0x1c;
buf[1] = 0x03;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x1c\n");
```

The following registers set up the display timing generator (DTG). The DTG is responsible for generating the timing signals going out to the display. Register 0x36 and 0x37 define the starting line number (the default value of 1 is used). Register 38 turns on the DTG in the 720P mode.

```
// Default value after reset
buf[0] = 0x36;
buf[1] = 0x00;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x36\n");
```

```
// Default value after reset (defines starting line number as 1)
buf[0] = 0x37;
buf[1] = 0x01;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x37\n");

// Turns on DTG and selects 720P SMPTE296M mode
buf[0] = 0x38;
buf[1] = 0x82;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x38\n");
```

The following registers set up the timing and window size information. Remember, these must coincide with what was programmed in the TMS320DM6446 side.

```
// Negative Hsync width (half of total width since there is corresponding positive synch width)
// Total width is 0x50 as defined per TMS320DM 6446 VENC_HSPLS
buf[0] = 0x25;
buf[1] = 0x28;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x25\n");

// positive hsync width (half of total width since there is corresponding negative synch width)
// Total width is 0x50 as defined per TMS320DM 6446 VENC_HSPLS register
buf[0] = 0x27;
buf[1] = 0x28;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x27\n");

//The following two registers defines the distance from sync to start of active video (0x12C)
//This was defined as 300 (0x12C) per TMS320DM6446 VENC_HSTART register

// LSBs of sync to active video [0:7]
buf[0] = 0x2a;
buf[1] = 0x2c;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x2a\n");

// MSB bit of sync to active video width: bit[6]
buf[0] = 0x2b;
buf[1] = 0xc0;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x2b\n");

// Distance from end of Active Video to start of negative sync; this is equivalent to the number
of blanking
// pixels to the right of the active window which is 70 (see section 1.4.1)
buf[0] = 0x26;
buf[1] = 0x46;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x26\n");

// The following two registers define total pixels per line (0x672 = 1650 per SMPTE296M)

// MSB of total pixels per line: bits[4:0]
buf[0] = 0x34;
buf[1] = 0x06;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x34\n");

// LSBs of total pixels per line: bits[7:0]
buf[0] = 0x35;
buf[1] = 0x72;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x35\n");
```

```
// The following two registers define total pixels per line (0x2ee = 750 per SMPTE296M)

// [6:4] are MSBs of lines per frame
buf[0] = 0x39;
buf[1] = 0x27;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x39\n");

// LSBs of lines per frame
buf[0] = 0x3a;
buf[1] = 0xee;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x3a\n");

// per THS8200, this must be set higher then lines per frame for progressive modes.
// MSBs are register 39 [2:0], hence we program this to the maximum 0x7ff
buf[0] = 0x3b;
buf[1] = 0xff;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x3b\n");
```

## A.4 1080i Firmware Configurations

This section discusses the registers that must be programmed on both the TMS320DM6446 and THS8200 parts to achieve the 1080i mode, including why you should program these registers. It focuses on the relevant registers, instead of going through the driver code line by line.

There are three main areas that must be programmed correctly to output 1080i video correctly. First, the OSD Window(s) on the DM6446 must be set appropriately per the SMPTE274M standard. Second, the VPBE VENC on the DM6446 must be programmed to generate the appropriate video timing and pixel clock information for the THS8200 device to use. And third, the THS8200 device must be programmed to accept the corresponding window size and timing information that it receives from the DM6446. In addition to these, there are a few other DM6446 registers that do not belong to the OSD or VENC hardware blocks that must be programmed. These include:

```
// Enable Field ID output since this is a interlaced video format
dispc_reg_merge(PINMUX0, PINMUX0_LFLDEN, PINMUX0_LFLDEN);

// Set PINMUX0 register to enable LCD output
dispc_reg_merge(PINMUX0, PINMUX0_LOEEN, PINMUX0_LOEEN);
```

### A.4.1 Video Window Settings

Similar to the 720P, this setting is primarily dictated by the SMPTE274M standard. Note that the DM6446 has four display windows, two video windows and two OSD windows; each of these four windows can be programmed independently. Therefore, when defining a new video format (such as 1080i), you must update the corresponding window sizes of all the display window(s) you plan on using. This is done using the following code segment:

```
#define BASEX1080I        x58         // 88 pixels
#define BASEY1080I        0x5         // 5 lines
#define DISP_XRES1080I    1920        // active horizontal pixels per SMPTE274M
#define DISP_YRES1080I    1088        // active vertical lines per SMPTE274M

// Set new baseX and baseY
dispc_reg_out(OSD_BASEPX, BASEX1080I);
dispc_reg_out(OSD_BASEPY, BASEY1080I);

// Set VID0 window  origin and size
dispc_reg_out(OSD_VIDWIN0XP, 200 - BASEX1080I);     // Actual offset is 200
dispc_reg_out(OSD_VIDWIN0YP, 13);                   // Actual offset is 13 + BASEY1080I = 18
dispc_reg_out(OSD_VIDWIN0XL, DISP_XRES1080I);
dispc_reg_out(OSD_VIDWIN0YL, DISP_YRES1080I/2);     // since 2 fields

// Set VID1 window  origin and size
dispc_reg_out(OSD_VIDWIN1XP, 200 - BASEX1080I);
```

```
dispc_reg_out(OSD_VIDWIN1YP, 13);
dispc_reg_out(OSD_VIDWIN1XL, DISP_XRES1080I);
dispc_reg_out(OSD_VIDWIN1YL, DISP_YRES1080I/2);

// Set OSD0 window  origin and size
dispc_reg_out(OSD_OSDWIN0XP, 200 - BASEX1080I);
dispc_reg_out(OSD_OSDWIN0YP, 13);
dispc_reg_out(OSD_OSDWIN0XL, DISP_XRES1080I);
dispc_reg_out(OSD_OSDWIN0YL, DISP_YRES1080I/2);

// Set OSD1 window  origin and size
dispc_reg_out(OSD_OSDWIN1XP, 200 - BASEX1080I);
dispc_reg_out(OSD_OSDWIN1YP, 13);
dispc_reg_out(OSD_OSDWIN1XL, DISP_XRES1080I);
dispc_reg_out(OSD_OSDWIN1YL, DISP_YRES1080I/2);
```

The SMPTE274M defines the vertical blanking on top of the active video as lines 1 through 21. The offset per the comments in the code segment above is actually 18 lines; it is often necessary to adjust this number slightly to avoid seeing invalid data at the beginning of the active video window. SMPTE274M also defines the total line length of a 1080i format as 2200 pixels, while the active video is only 1920. This leaves 280 horizontal blanking pixels that must be split between the left and right side of the 1920 active video pixels. SMPTE274M is flexible in how these horizontal blanking pixels are split and the choice was made to split them as 200 on the left side (leaving 80 on the right side) of the active video. Note the choice that was made must be reflected on the THS8200 side, as seen in later sections.

Finally, the windows must be enabled; for this application, the choice was made to display the 1080i video in video window 0 due to the buffer bandwidth limitations associated with video window 1.The OSD field inversion bit is normally set when using video window 1 as a workaround for the silicon bug described in the *TMS320DM6446 Digital Media System-on-Chip Silicon Errata* (SPRZ241). This is not necessary in video window 0; therefore, this bit will be cleared. The following code segment enables video window 0, sets it in frame mode (for interlaced video), and clears the field inversion bit.

```
// Enable Video Window 0 / disable video window 1
dispc_reg_out(OSD_VIDWINMD, OSD_203);

// Clear OSD Field Inversion for VID0 Use
dispc_reg_out(OSD_MODE, 0);
```

### A.4.2    VPBE VENC Register Configurations

This section discusses how to program the VENC so that it defines the appropriate timing for the video format the program is going to output. This strictly follows the SMPTE274M video standard and the horizontal blanking interval used when defining the window sizes.

```
// Set Timing parameters for 720P frame (must match what THS8200 expects)
dispc_reg_out(VENC_HSPLS, BASEX1080I);
dispc_reg_out(VENC_VSPLS, BASEY1080I);
dispc_reg_out(VENC_HINT, 2200-1);
dispc_reg_out(VENC_HSTART, 200);
dispc_reg_out(VENC_HVALID, DISP_XRES1080I);
dispc_reg_out(VENC_VINT, 1125-1);
dispc_reg_out(VENC_VSTART, 13);
dispc_reg_out(VENC_VVALID, DISP_YRES1080I/2);
dispc_reg_out(VENC_HSDLY, 0);
dispc_reg_out(VENC_VSDLY, 0);
dispc_reg_out(VENC_YCCCTL, 0);
dispc_reg_out(VENC_VSTARTA, 13);
```

The timing defined above generates the horizontal and vertical syncs output to the THS8200. Another signal that the DM6446 outputs to the THS8200 is the pixel clock (VLCK in Figure 12). To accommodate 1080i, which requires a pixel clock of 74.25 per SMPTE274M, an external clock is needed. Per the DVEVM design, DM6446 only has a 27 MHz clock used by the VENC internally; this is sufficient for NTSC video format, but not for 1080i. The clock output to the THS8200, VLCK, is dependent on the external clock input to the DM6446 via the VPBECLK pin and the DCLK register settings. The DLCK registers on the TMS320DM6446 hardware component can be configured to use the clock input into VPBECLK pin and generate a clock pattern up to 64 cycles long; however, for the purposes of the solution being presented in this application report, the DLCK registers are configured to generate a clock equal to the input clock at the VPBECLK pin. The clock generated via the DLCK registers is output via the VCLK pin on the TMS320DM6446 and is used to drive the THS8200. The register settings are as follows:

```
// Enable DCLOCK
dispc_reg_out(VENC_DCLKCTL, VENC_DCKCTL_DCKEC);

// Set DCLOCK pattern
dispc_reg_out(VENC_DCLKPTN0, 1);
dispc_reg_out(VENC_DCLKPTN1, 0);
dispc_reg_out(VENC_DCLKPTN2, 0);
dispc_reg_out(VENC_DCLKPTN3, 0);
dispc_reg_out(VENC_DCLKPTN0A, 2);
dispc_reg_out(VENC_DCLKPTN1A, 0);
dispc_reg_out(VENC_DCLKPTN2A, 0);
dispc_reg_out(VENC_DCLKPTN3A, 0);
dispc_reg_out(VENC_DCLKHS, 0);
dispc_reg_out(VENC_DCLKHSA, 1);
dispc_reg_out(VENC_DCLKHR, 0);
dispc_reg_out(VENC_DCLKVS, 0);
dispc_reg_out(VENC_DCLKVR, 0);
```

The digital LCD output is then enabled by setting the following register. You must also enable output of horizontal and vertical syncs, as well as program VENC to use the external VPBE CLK and enable the VENC to start outputting video and timing signals to be fed to THS8200. These steps are shown in the code segment below.

The following code enables several processes:

- Enables the digital LCD output
- Enables output of horizontal and vertical syncs
- Programs the VENC to use the external VPBE CLK and enable the VENC to start outputting video and timing signals to be fed to THS8200

These steps are shown in the code segment below:

```
// Enable LCD output control
dispc_reg_out(VENC_LCDOUT, 0x1);

// turning on horizontal and vertical syncs
```

```
dispc_reg_out(VENC_SYNCCTL, (VENC_SYNCCTL_SYEV|VENC_SYNCCTL_SYEH) );

// Enable external VPBE CLK to VENC
dispc_reg_out(VPSS_CLKCTL, 0x0a);

// Enable all VENC, non-standard timing mode, master timing, HD, interlaced
dispc_reg_out(VENC_VMOD, (VMOD_VENC|VMOD_VMD | VMOD_HDMD | VMOD_NSIT) )
```

### A.4.3 THS8200 Register Configurations

The details of the THS8200 are documented in the code segment below. The register field definitions are not covered in this document; for details of the register fields for each individual register see *THS8200 All-Format Oversampled Component Video/PC Graphics D/A System With Three 11-Bit DACs, CGMS Data Insertion Data Manual* (SLES032). Note that buf[0] contains the register address and buf[1] the register data, both of which are sent across the I2C bus.

The following registers set up the input and output modes.

```
// Sets YPbPr output mode, and sets polarity of the horizontal and vertical sync signals
buf[0] = 0x82;
buf[1] = 0x1f;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x82\n");

// Accept 20-bit YCbCr data
buf[0] = 0x1c;
buf[1] = 0x03;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x1c\n");
```

Registers 0x36 and 0x37 define the starting line number (the default value of 1 was used). Register 38 turns on the DTG in 1080i mode.

```
// Default value after reset
buf[0] = 0x36;
buf[1] = 0x00;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x36\n");

// Default value after reset (defines starting line number as 1)
buf[0] = 0x37;
buf[1] = 0x01;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x37\n");

// Turns on DTG and selects 1080i SMPTE274M mode
buf[0] = 0x38;
buf[1] = 0x81;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x38\n");
```

The following registers set up timing and window size information. Remember, these must coincide with what was programmed in the DM6446 side.

```
// Negative Hsync width (half of total width since there is corresponding positive synch width)
// Total width is 0x58 as defined per TMS320DM 6446 VENC_HSPLS
buf[0] = 0x25;
buf[1] = 0x2C;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x25\n");

// positive hsync width (half of total width since there is corresponding negative synch width)
// Total width is 0x50 as defined per TMS320DM 6446 VENC_HSPLS register
buf[0] = 0x27;
buf[1] = 0x2C;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x27\n");
```

```
//The following two registers defines the distance from sync to start of active video (0xC8)
//This was defined as 200 (0xC8) per TMS320DM6446 VENC_HSTART register

// LSBs of sync to active video [0:7]
buf[0] = 0x2a;
buf[1] = 0xC8;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x2a\n");

// MSB bit of sync to active video width: bit[6]
buf[0] = 0x2b;
buf[1] = 0x00;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x2b\n");

// Distance from end of Active Video to start of negative sync; this is equivalent to the number
of blanking
// pixels to the right of the active window which is 70 (see section 1.4.1)
buf[0] = 0x26;
buf[1] = 0x50;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x26\n");

// The following two registers define total pixels per line (0x898 = 2200 per SMPTE274M)

// MSB of total pixels per line: bits[4:0]
buf[0] = 0x34;
buf[1] = 0x08;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x34\n");

// LSBs of total pixels per line: bits[7:0]
buf[0] = 0x35;
buf[1] = 0x98;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x35\n");

// The following two registers define total pixels per line (0x465 = 1125 per SMPTE274M)

// [6:4] are MSBs of lines per frame
buf[0] = 0x39;
buf[1] = 0x42;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x39\n");

// LSBs of lines per frame
buf[0] = 0x3a;
buf[1] = 0x65;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x3a\n");

// lines per field; MSBs are register 39 [2:0], LSBs are reg 3b[7:0]; (0x233)
// This should be set to 563 (0x233) per SMPTE274M
buf[0] = 0x3b;
buf[1] = 0x233;
if(write(i2cfd, buf, 2) != 0)
   printk("Write Error Address = 0x3b\n");
```

## A.5 LCD Firmware Configurations

### A.5.1 Video Window Settings

Because an LCD runs in non-interlaced mode, the video windows must be set up to run in field mode. Even though the LCD is not interlaced, the VPBE is set to output in field mode, so that the window sizes can be set with the correct resolution of the display. If the VPBE is set to frame mode, the window sizes have to be set to half of the actual resolution by setting VIDWINMD.VFF0, VIDWINMD.VFF1, OSDWIN0MD.OFF0, and OSDWIN1MD.OFF1 to 0. This process is discussed in more detail in Section A.5.4.

Other than that, the setting of window offsets and the viewable area are the same as the other display. Figure A-1 shows the different windows settings.
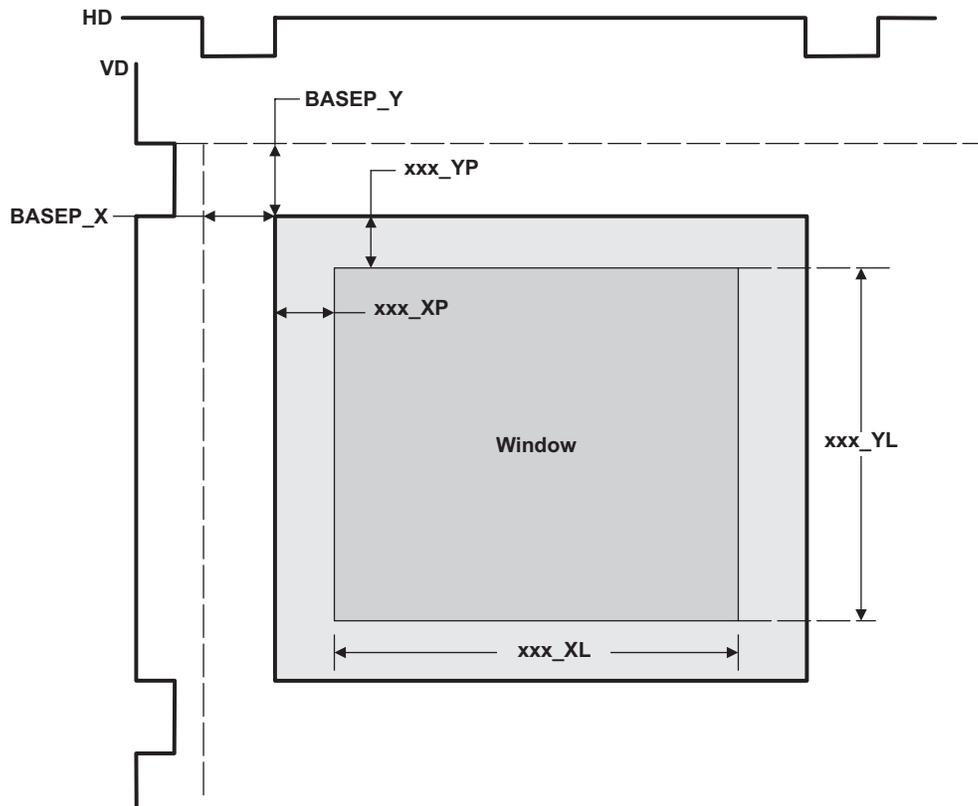


**Figure A-1. OSD Window Positioning**

For the Sharp LQ64D343 LCD, these registers are set as follows (in decimal):
- BASEP_X = 34
- BASEP_Y = 0
- xxx_XL = 640
- xxx_YL = 480

### A.5.2 VPBE VENC Register Configurations

This section discusses the process of setting up the VENC of VPBE for output to an LCD in RGB format.

TI suggests holding VENC in reset by setting VENC_VMOD = 0 whenever you are modifying the VENC registers.

All of the IO pins should be set up so that all multiplexed pins are configured. First, the RGB666 functionality is enabled on GPIO[46:47] by setting PINMUX0.RGB666 = 1. Next, the OE functionality is enabled on GPIO[0] by setting PINMUX0.LCD_OE = 1.

The timing settings for the LCD should also be set. These settings vary per LCD, and are determined by the timings provided in the LCD manufacturer's documentation. Figure A-2 shows the registers that must be set.
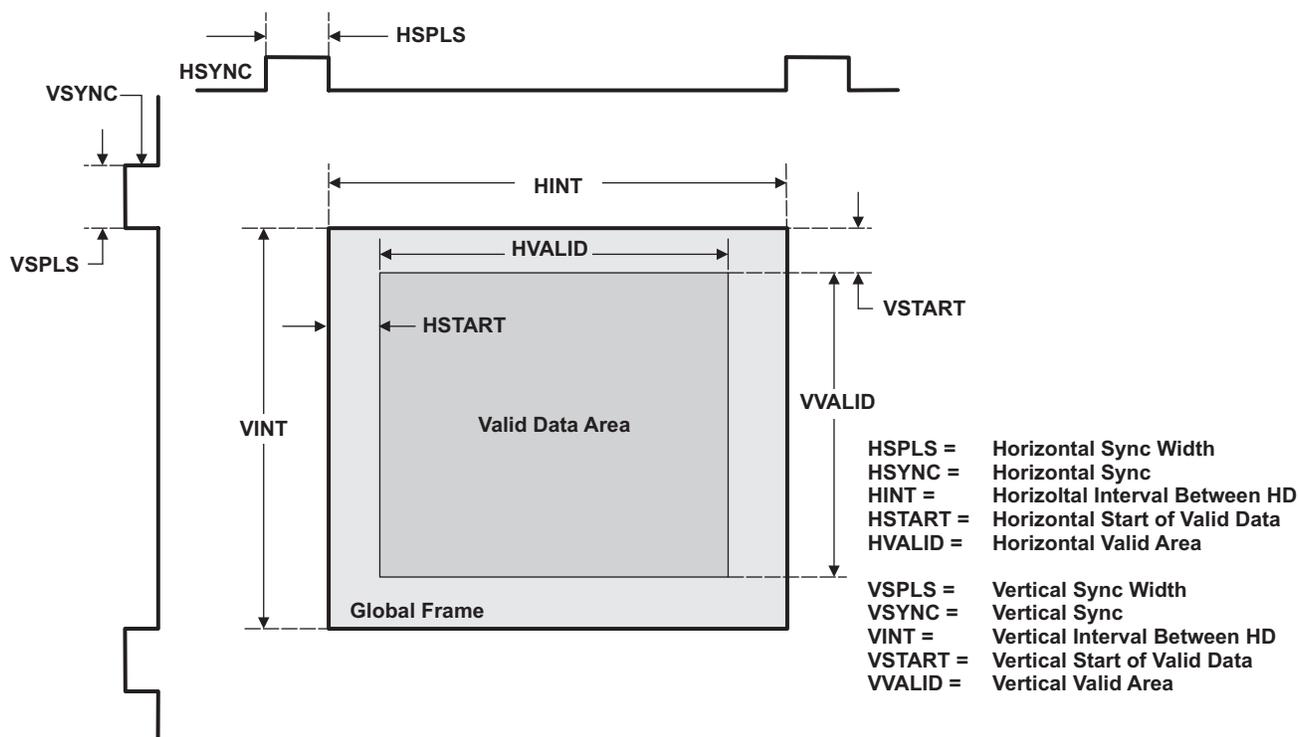


**Figure A-2. VENC Control Signal Definitions**

For the Sharp LQ64D343 LCD, these registers are set as follows (in decimal):
- HSPLS = 96
- VSPLS = 32
- HINT = 800
- HSTART = 23
- HVALID = 640
- VINT = 524
- VSTART = 32
- VVALID = 480

The VENC is set to follow these timings by setting VMOD.VMD = 1, otherwise the VENC runs at NTSC/PAL timings.

The HSYNC, VSYNC, and OE outputs are enabled by setting SYNCCTL.SYEH = 1, SYNCCTL.SYEV = 1, and LCDOUT.OEE = 1, respectively. The polarities for HSYNC, VSYNC, and OE are set with SYNCCTL.HPL, SYNCCTL.VPL, LCDOUT.OEP, respectively.

The pixel clock must be configured with the DCLKCTL and the associated DCLKPTN registers. These registers are configured with the following values:
- DCLKCTL = 0x801
- DCLKPTN0 = 0x3
- DCLKPTN1 = 0x0
- DCLKPTN2 = 0x0
- DCLKPTN3 = 0x0
- DCLKPTN0A = 0x3
- DCLKPTN1A = 0x0

- DCLKPTN2A = 0x0
- DCLKPTN3A = 0x0

The VCLK output is enabled by setting VIDCTL.VCLKE = 1. In addition, VPSS_CLKCTL.VENCCLKEN = 1, which enables the clock in the video encoder.

Finally, you must pull the VENC out of reset with it configured for RGB output. To set the VENC for RGB output, set VMOD.VDMD = 2. You must set VMOD.VMD = 1 to specify that the VENC should use the custom timing settings described previously. To pull the VENC out of reset, set VMOD.VENC = 1.

### A.5.3    Color Space Conversion

Conversion from YCbCr color space to RGB color space occurs in the VPBE RGB output formatter. This is activated automatically when the output is set for RGB mode. You have the option of specifying the coefficients of the conversion matrix; however, these are preloaded with default coefficients to output ITU-R BT601 levels of color data.

Figure A-3 shows a block diagram of how this color space conversion is done. In the first stage of the block diagram, the YCC inputs, which normally range from 0-255, are attenuated to ITU-R BT601 levels (Y:0-219, C:-128-128). This attenuation is enabled by setting the VDPRO.ATRGB = 1.
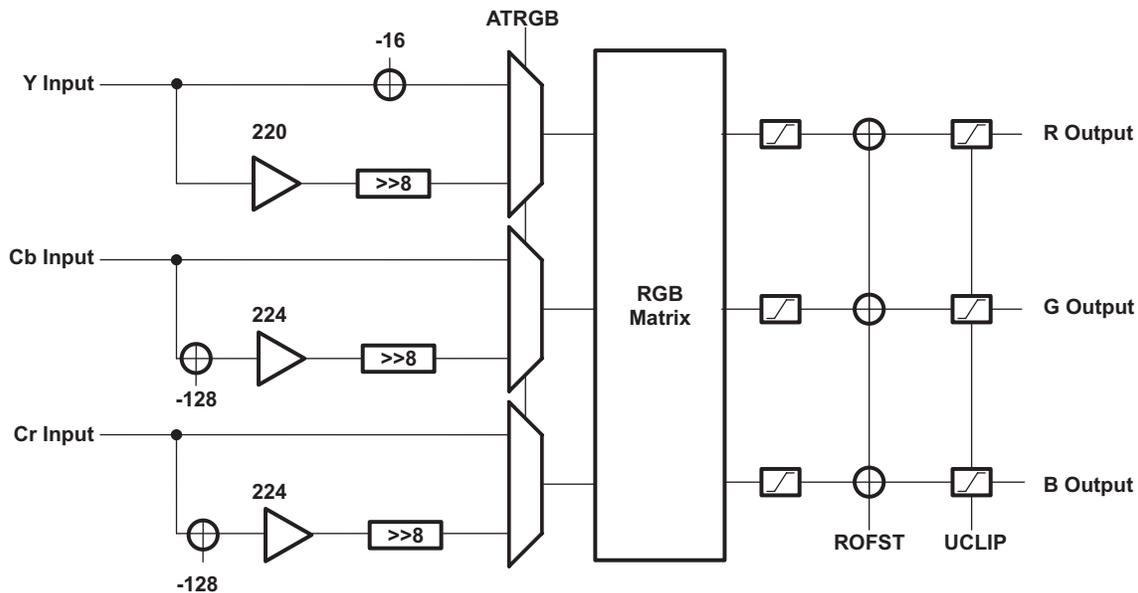


**Figure A-3. RGB Conversion Block Diagram**

The YCbCr color data is converted to RGB space in the RGB matrix using the following equation shown in Figure A-4:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \frac{1}{1024} \begin{bmatrix} GY & 0 & RV \\ GY & -GU & -GV \\ GY & BU & 0 \end{bmatrix} \begin{bmatrix} Y - 16 \\ Cb - 128 \\ Cr - 128 \end{bmatrix}$$

**Figure A-4. YCbCr to RGB Color Space Conversion**

The coefficients in the conversion matrix above can be set at runtime with registers DRGBX0-DRGBX4. These register have the default value of the ITU-R BT601 RGB conversion matrix.

Then the offset specified by RGBCLP.OFST is added, followed by the upper level clipping. The clip level is set by RGBCLP.UCLIP. The output RGB samples are limited to 8-bit resolution per color output.

### A.5.4 FBDev Driver Additions

The davincifb.c file requires some modifications to accommodate the addition of the LCD device and properly configure the VPBE. Here are the prototypes for the helper functions that were added to enable the use of a digital LCD.

```
static void davincifb_lcd_rgb_config(int on)
static void set_lcd_timings(int h_valid, int v_valid,
                                      int h_int,   int v_int,
                                      int h_pulse, int v_pulse,
                                      int h_start, int v_start,
                                      int h_dly,   int v_dly,
                                      int hsync_pol,
                                      int vsync_pol,
                                      int oe_pol)
struct lcd_timing_params
{
                u_int32_t h_interval;
                u_int32_t v_interval;
                u_int32_t h_valid;
                u_int32_t v_valid;
                u_int32_t h_pulse;
                u_int32_t v_pulse;
                u_int32_t h_start;
                u_int32_t v_start;
                u_int32_t h_delay;
                u_int32_t v_delay;
                u_int32_t hsync_pol;  // 0 = active low, 1 = active high
                u_int32_t vsync_pol;  // 0 = active low, 1 = active high
                u_int32_t oe_pol;     // 0 = active low, 1 = active high
```

Now that there is a function that sets up the VPBE as needed to use the LCD, some routines must be set up to call it. First, you set up support for the boot-time arguments by adding the parseable boot arguments to the davincifb_setup function. In this function, you must add a case for the LCD under the output parser and an RGB case under the format parser. Sections were added to the code below.

```
int __init davincifb_setup(char *options)

                        if (!strncmp(this_opt, "output=", 7)) {
                                        else if (!strncmp(this_opt + 7, "lcd", 3))
                                                        dmparams.output = LCD;

                        else if (!strncmp(this_opt, "format=", 7))
                        else if (!strncmp(this_opt + 7, "rgb", 3))
                                                        dmparams.format = RGB;
```

Now a call must be added to the config routine whenever these boot arguments are passed by adding a case to davincifb_probe that checks for an LCD output in the RGB format. When this is the case, you call the davincifb_lcd_rgb_config function, which configures the VPBE to use the LCD as the display device.

```
static int davincifb_probe(struct device *dev){

                        else if((dmparams.output == LCD)&&(dmparams.format == RGB))
                                        dm->output_device_config = davincifb_lcd_rgb_config;
                        /* Add support for other displays here */
```

When you call the config function, the default timing parameters are passed to it; these default parameters are defined in a static structure called default_lcd_timing, which is defined near the top of davincifb.c.

```
// default LCD timing parameters
static struct lcd_timing_params default_lcd_timing = {
                .h_interval = 800, // number of pixel clocks between active hsync edges
                .v_interval = 524, // number of hsyncs  between active vsync edges
                .h_valid   = 640, // x resolution
                .v_valid   = 480, // y resolution
                .h_pulse   = 96,  // width in pixel clocks of hsync
                .v_pulse   = 32,  // width in hsync pulses  of vsync
                .h_start   = 23,  // time in pixel clocks from active hsync to valid data
                .v_start   = 32,  // time in hsync pulses from active vsync to new frame
```

```
                    .h_delay    = 0,
                    .v_delay    = 0,
                    .hsync_pol  = 0,   // 0 = active low, 1 = active high
                    .vsync_pol  = 0,   // 0 = active low, 1 = active high
                    .oe_pol     = 1,   // 0 = active low, 1 = active high
};
```

The following #defines were added near the top of davincifb.c:

```
static int davinc#define BASEXVGA       0x0
#define BASEYVGA                       0x22
#define DISP_XRESVGA                   640
#define DISP_YRESVGA                   480
            #define LCD_PANEL_CLOCK  37037    // 27 MHz Pixel Clock
```

To enable changing the LCD timing parameters at runtime, an IOCTL command called FBIO_SETLCDTIMEPARAMS was added to davinci_ioctl. This function calls the set_lcd_timings function with the data contained in a lcd_timing_params structure that is then passed into the ioctl function call. This function call allows you to change the LCD timing parameters at runtime.

```
...
struct lcd_timing_params lcd_timing;
...
                case FBIO_SETLCDTIMEPARAMS:
                        if (copy_from_user(&lcd_timing,argp,sizeof(lcd_timing)))
                                        return -EFAULT;
                        if (lcd_timing.h_valid    >  0 &&
                            lcd_timing.v_valid    >  0 &&
                            lcd_timing.h_interval >  0 && lcd_timing.v_interval >  0 &&
                            lcd_timing.h_pulse    >  0 &&
                            lcd_timing.v_pulse    >  0 &&
                            lcd_timing.h_start    >= 0 &&
                            lcd_timing.v_start    >= 0 &&
                            lcd_timing.h_delay    >= 0 &&
                            lcd_timing.v_delay    >= 0 &&
                            lcd_timing.h_valid < lcd_timing.h_interval &&
                            lcd_timing.v_valid < lcd_timing.v_interval ){

                        set_lcd_timings(lcd_timing.h_valid,
                                        lcd_timing.v_valid,
                                        lcd_timing.h_interval,
                                        lcd_timing.v_interval,
                                        lcd_timing.h_pulse,   lcd_timing.v_pulse,
                                        lcd_timing.h_start,   lcd_timing.v_start,
                                        lcd_timing.h_delay,   lcd_timing.v_delay,
                                        lcd_timing.hsync_pol,
                                        lcd_timing.vsync_pol,
                                        lcd_timing.oe_pol);
                                        return 0;
                        } else {
                                        return -EINVAL;
                        }
                             break;
```

Initially the driver was only prepared to handle interlaced displays. To support non-interlaced displays, you have to add some cases to disable interlacing in the functions davincifb_set_par and init_fb_info. The modifications are shown in the code below.

```
static int davincifb_set_par(struct fb_info *info)
                ...
                // ---- start of original code ----
                //set_interlaced(info->fix.id, 1);
                //set_win_position(info->fix.id,
                //                   x_pos(w), y_pos(w), v->xres, v->yres / 2);
                // ----- end of original code -----
                if(dmparams.output == LCD && dmparams.format == RGB){
                                // non interlaced output
                                set_interlaced(info->fix.id, 0);
```

```
                                        set_win_position(info->fix.id,
                                            x_pos(w), y_pos(w), v->xres, v->yres);
                }
                else {
                                // interlaced output
                                set_interlaced(info->fix.id, 1);
                                set_win_position(info->fix.id,
                                        x_pos(w), y_pos(w), v->xres, v->yres / 2);
                }
                ...
static struct fb_info *init_fb_info(struct dm_win_info *w,
                struct fb_var_screeninfo *var, char *id)
                ...
                if(dmparams.output == LCD && dmparams.format == RGB){
                                info->fix.type_aux = 1; // no interlacing
                }
                else{
                                // originally only this code was here
                                info->fix.type_aux = 0; // interlacing
                }...
```

With these modifications to the driver, the LCD can be used as a framebuffer device hosting framebuffers 0-3 within Linux.

In addition, functionality was added to allow runtime switching between a digital LCD, NTSC, PAL, 720p, and 1080i displays. To allow for this, the davincifb_set_par() function must be modified. The driver detects which device to use based on the resolution of the frame it receives. For the digital LCD to be realized, the 640×480 pixel frame functionality has to be used.

To enable this functionality, you must add some code to the beginning and end of the davincifb_set_par() function that detects the resolution of the display and configures the VENC to properly handle the corresponding display. The following code detects resolution; this is located as the first code to be executed in davincifb_set_par():

```
static int davincifb_set_par(struct fb_info *info)
{
                struct dm_win_info *w = (struct dm_win_info *)info->par;
                struct fb_var_screeninfo *v = &info->var;
                u32 start = 0, offset = 0;
                int bufs = 0;

                DBGENTER;

                // pre-processing to determine if a valid video format is requested;
        // otherwise, defualting to NTSC
        if ((v->xres == DISP_XRESD1 ) && (v->yres == DISP_YRESD1))
        {
          dmparams.output = NTSC;
          v->vmode = FB_VMODE_INTERLACED;
          printk("FBDEV: Configuring for NTSC\n");
        }
        else if ((v->xres == DISP_XRESD1 ) && (v->yres == 576))
        {
          dmparams.output = PAL;
          v->vmode = FB_VMODE_INTERLACED;
          printk("FBDEV: Configuring for PAL\n");
        }
        else if ((v->xres == DISP_XRES720P ) && (v->yres == DISP_YRES720P))
        {
          dmparams.output = LCD_720P;
          v->vmode = FB_VMODE_NONINTERLACED;
          printk("FBDEV: Configuring for 720P\n");
        }
        else if ((v->xres == DISP_XRES1080I ) &&
                                ((v->yres == DISP_YRES1080I) || (v->yres == 1080)))
        {
          dmparams.output = LCD_1080I;
```

```
                        v->vmode = FB_VMODE_INTERLACED;
                        printk("FBDEV: Configuring for 1080I\n");
                      }
                      else if(((dmparams.output == LCD) && (dmparams.format == RGB))
                       || ((v->xres == 640) && (v->yres == 480))) // allow runtime switching
                      {
                        dmparams.output = LCD;
                        dmparams.format = RGB;
                        v->vmode = FB_VMODE_NONINTERLACED;
                        printk("FBDEV: Configuring for LCD\n");
                      }
                      else
                      {
                        printk("Unrecognized output format (defaulting to NTSC)\n");
                        dmparams.output = NTSC;
                        v->vmode = FB_VMODE_INTERLACED;
                        v->xres = 720;
                        v->yres = 480;
                      }

                      // JG: Fix virtual x and y settings for the requested resolution
                      if (is_win(info->fix.id, VID0) || is_win(info->fix.id, VID1))
                      {
                        bufs = TRIPLE_BUF;
                      }
                      else
                      {
                        bufs = DOUBLE_BUF;
                      }

                      fix_default_var(w, v->xres, v->yres, v->xoffset, v->yoffset, bufs);


                      //JG:  ----------------  START of original code ------------------------

                      info->fix.line_length = v->xres_virtual * v->bits_per_pixel / 8;
```

When the VGA resolution or the boot arguments are in LCD format, the LCD is used as an output. Note that the LCD is the lowest priority display and only used when all other resolutions fail. The resolution of the input is 640×480 *or* the boot arguments call out the LCD.

Now that resolutions are being detected while the program is running and other progressive scan displays are being used, there is a more elegant way to set the interlacing portions of this function. The code was originally written as follows:

```
static int davincifb_set_par(struct fb_info *info)
                ...
                // ---- start of original code ----
                //set_interlaced(info->fix.id, 1);
                //set_win_position(info->fix.id,
                //                 x_pos(w), y_pos(w), v->xres, v->yres / 2);
                // ----- end of original code -----
                if(dmparams.output == LCD && dmparams.format == RGB){
                                // non interlaced output
                                set_interlaced(info->fix.id, 0);
                                        set_win_position(info->fix.id,
                                        x_pos(w), y_pos(w), v->xres, v->yres);
                }
                else {
                                // interlaced output
                                set_interlaced(info->fix.id, 1);
                                set_win_position(info->fix.id,
                                        x_pos(w), y_pos(w), v->xres, v->yres /
2);
                }
                ...
```

Below is the code with the the more elegant solution:

```
static int davincifb_set_par(struct fb_info *info)
                ...
                if (v->vmode == FB_VMODE_INTERLACED)
                {
                            set_interlaced(info->fix.id, 1);
                set_win_position(info->fix.id, x_pos(w), y_pos(w), v->xres, v->yres / 2);
                }
                else
                {
                            set_interlaced(info->fix.id, 0);
                set_win_position(info->fix.id, x_pos(w), y_pos(w), v->xres, v->yres);
                }
                ...
```

When the resolution detection occurs, the v->mode value that was set earlier determines whether the display is interlaced or not.

To allow runtime switching, you must now reconfigure the output by calling the config function for the corresponding display. The code to allow this is shown below:

```
static int davincifb_set_par(struct fb_info *info)
        ...

        set_win_enable(info->fix.id, 1);

        //JG:  ---- END of Original Code/ Beginning of our Code ---//

        if (dmparams.output == NTSC )
        {
                davincifb_ntsc_composite_config(1);
        }
        else if (dmparams.output == PAL )
        {
                davincifb_pal_composite_config(1);
        }
        else if(dmparams.output == LCD && dmparams.format == RGB)
        {
                davincifb_lcd_rgb_config(1);
        }
        else if (dmparams.output == LCD_720P)
        {
           davincifb_720p_config(1);
           configureVideoDAC(LCD_720P);
        }
        else if (dmparams.output == LCD_1080I)
        {
           davincifb_1080i_config(1);
           configureVideoDAC(LCD_1080I);
        }


        RETURN(0);
}
```

One remaining step should be done to complete the runtime switching enable. The config functions are now called from davincifb_set_par(), so they no longer need to be called from davincifb_probe(). The following code shows how to disable this call comment:

```
static int davincifb_probe(struct device *dev)
…
                    /* Turn ON the output device */
//                  dm->output_device_config(1);

                    RETURN(0);

        exit:
                    davincifb_remove(dev);

        iounmap((void *)dm->mmio_base);
        release_mmio:
                     release_mem_region(dm->mmio_base_phys, dm->mmio_size);

                    RETURN(-ENODEV);
}
```

# IMPORTANT NOTICE